An Investigation of Passive Sonar on an AUV

by

Heinrich Wesson

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering in the Faculty of Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Mr. W. Smit
Department of Electrical and Electronic Engineering

March 2017

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Heinrich Wesson

Date: March 2017

Abstract

This thesis provides a theoretical foundation for a passive sonar system with the focus being on delay-and-sum beamforming. The parallel nature of the delay-and-sum beamforming algorithm is exploited by using NVIDIA's Compute Unified Device Architecture (CUDA) for Graphics Processing Units (GPUs). Theory and simulations are used to determine the feasibility of such a system on an AUV.

The theory of each element that forms part of a passive sonar system is discussed. Similarly, CUDA is also discussed with a focus on theory that was used in the beamformer simulations.

Homogeneous and isotropic ambient noise in the sea and a submarine's signal at periscope depth is simulated.

Using these signals, near-field and far-field delay-and-sum beamforming simulations in two and three dimensions are done in MATLAB and CUDA C. Meaningful results were obtained using planar and volumetric arrays with four, eight, and nine sensors.

Increasing the number of synchronous beams via increasing the temporal sampling frequency had significant beneficial effects of estimating target position or bearing. Resilience against decreasing signal-to-noise ratio was also shown to improve.

Promising results were obtained when implementing the delay-and-sum beamforming algorithm in CUDA C. A comparison to standard C yielded an 11.67-fold decrease in execution time at best. The use of pinned Central Processing Unit (CPU) memory and CUDA streams facilitated concurrent execution of memory copies and kernel executions. Concurrent execution of kernels were also achieved.

Many applications of passive sonar require large arrays that is not feasible for application on an AUV. A retractable array can alleviate this problem, also enabling the array to control spacing between sensors and thus operate at multiple frequencies.

Harbours and naval bases can be vulnerable to an underwater attack from divers. Securing these perimeters with fences can become a complex task. Concluded from the research in this thesis, a diver detection sonar is the most feasible application of a passive sonar system on an AUV. The absence of the need for long distance near-field detection and the need for short spacing between sensors in the array counts in favour of this application.

Opsomming

Hierdie tesis bevat ‘n teoretiese grondslag vir ‘n passiewe sonar sisteem met spesifieke fokus op die vertraag-en-sommeer straalvormingsalgoritme. NVIDIA se “Compute Unified Device Architecture” (CUDA) vir grafiese verwerkingseenhede word gebruik om te kapitaliseer op die parallelle natuur van hierdie algoritme. Teorie en simulasies is gebruik om die lewensvatbaarheid van so ‘n stelsel te bepaal op ‘n outonome onderwater voertuig (OOV).

Elke element wat deel vorm van ‘n passiewe sonar sisteem is bespreek uit ‘n teoretiese oogpunt. Soortgelyk word CUDA bespreek met ‘n fokus op teorie wat toepaslik is op straalvormings-simulasies in hierdie tesis.

Homogene en isotropiese omgewingsgeraas in die see asook ‘n duikboot se sein by perikoop diepte is gesimuleer.

Deur hierdie seine te gebruik word naby-veld en ver-veld vertraag en sommeer straalvormings simulasies gedoen in twee- en drie dimensies, deur gebruik te maak van MATLAB en CUDA C. Betekenisvolle resultate was verkry met platvlak- en volumetriese skikkings wat vier, agt en nege sensors bevat het.

‘n Vermeerdering in die aantal sinkroniese strale deur die monster frekwensie in tyd te vermeerder het beduidende voordelige effekte gehad met betrekking tot die bepaling van teiken posisie of peiling. Veerkragtigheid teen ‘n vermindering van sein-tot-ruis verhourhouding het ook verbeter.

Belowende resultate was verkry in CUDA C met die implementering van die vertraag-en-sommeer straalvormingsalgoritme. ‘n Vergelyking met standaard C het in die beste geval ‘n 11.67-foud vermindering in uitvoeringstyd gelewer. Die gebruik van vasgepende sentrale verwerkingseendheid geheue en CUDA “streams” het konkurrente uitvoering van geheue kopieë en CUDA “kernel” uitvoering moontlik gemaak.

Meeste toepassings van passiewe sonar vereis groot skikkings wat dit nie moontlik maak vir toepassing op ‘n OOV nie. ‘n Terugtrekbare skikking kan hierdie probleem minder prominent maak en terselfdetyd ook die vermoë gee om by verskillende frekwensies te kan werk.

Hawens en vloot basisse is kwesbaar tot onderwater-duiker aanvalle. Dit is ‘n komplekse taak om hierdie areas met onderwater-heinings te beveilig. Aangesien langafstand naby-veld deteksie nie nodig is nie en kort spasieering tussen sonsors vereis word in die skikking, is duiker deteksie die mees vatbare toepassing van ‘n passiewe sonar sisteem op ‘n OOV.

Acknowledgements

I would like to take this opportunity to express my gratitude to the following people and institutions:

- **My parents**, for their unconditional love, financial and emotional support, and encouragement throughout the execution of this project. Without the perseverance and determination instilled by them I would not have been able to come this far.
- **Mr Willem Smit**, my supervisor, for his ongoing support, encouragement through difficult times, and always believing in me.
- **Talita Beyl**, for her unconditional love, supporting me through difficult times, and helping me with proof reading and editing of this thesis.
- **Prof Thomas Niesler**, for his help with signal simulations. Thanks for always being kind and willing to help with questions related to digital signal processing.
- **Mr. Philip La Grange** at the Institute for Maritime Technology (IMT), for his insight into passive sonar systems.
- **IMT**, for providing me with financial support to carry out this project.

Table of Contents

List of Figures

List of Tables

Nomenclature

2D	Two Dimensional
3D	Three Dimensional
ADC	Analog to Digital Converter
AGC	Automatic Gain Controller
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDS	Diver Detection Sonar
DI	Directivity Index
DT	Detection Threshold
FIR	Finite Impulse Response
GPU	Graphics Processing Unit
IIR	Infinite Impulse Response
LOFAR	Low Frequency and Analysis Recording
MP	Multiprocessor
NL	Noise Level
PSD	Power Spectral Density
SL	Source Level
SM	Streaming Multiprocessor
SNR	Signal-to-Noise Ratio
SONAR	Sound Navigation And Ranging
SQNR	Signal to Quantisation Noise Error
TL	Transmission Loss

Chapter 1
Introduction

1.1 Background

Compared to other forms of radiation, sound waves travel immense distances in water [1]. Even noted as long ago as the year 1490 by the famous engineer Leonardo da Vinci, he stated [1]: "If you cause your ship to stop and place the head of a long tube in the water and place the outer extremity to your ear, you will hear ships at a great distance from you." This fact provides the platform for what we know today as SOund NAvigation and Ranging, or in short, SONAR.

A sonar system is defined as a system that uses acoustic energy to communicate with- or observe any target of interest in water [2]. Two main categories of sonar exist, namely active and passive sonar. Active sonar consists of a projector that sends out a pre-defined acoustic signal into the water and a receiver that receives the echo of that projected signal reflected by a target. Echo return time and direction of arrival are then measured and used to detect and locate targets. Passive sonar on the other hand only receives signals and does not have a projector. Rather, to calculate the direction of arrival, it relies on the noise that the target itself emanates such as the engine noise of a marine vessel. The process of using the spatial directivity of a receiver or an array, brought on by the spatial separation of the elements in the array, to filter spatial information and obtain the direction of arrival of a target's noise (passive sonar) or reflected signal (active sonar) is known as beamforming [3].

There exists a wide variety of applications of sonar systems in public, scientific and military settings. Examples of these include the active acoustic detection and tracking of fish (fish finder) for recreational or commercial fishing purposes, the passive acoustic detection of whales for ecological surveying purposes [4], and the active or passive detection and tracking of marine vessels for military or surveillance purposes.

1.2 Problem Statement

This open-ended thesis focusses on the investigation and simulation of the theory governing a passive sonar system. From the knowledge obtained, the feasibility of such a system for an Autonomous Underwater Vehicle (AUV) is determined.

1.3 Objectives of this thesis

Seeing that this thesis is the first of its kind at the University of Stellenbosch, the main objective of this thesis is to study the theory behind passive sonar systems with a focus on the delay-and-sum beamforming algorithm. The delay-and-sum beamforming algorithm is simulated in two and three dimensions in MATLAB for theory investigation and also in CUDA C to exploit the parallelism of the delay-and-sum beamforming algorithm. With the knowledge obtained

through theory and simulation, the feasibility of the use of a passive sonar system on an AUV is determined.

1.4 Overview of thesis

The remaining chapters in this thesis contain the following:

- **Chapter 2** serves as an overview of basic terminology, theory, and hardware that accompanies passive sonar systems.
- **Chapter 3** gives an overview of CUDA theory that is used to implement the delay-and-sum beamforming algorithm in CUDA C.
- **Chapter 4** includes a discussion of the delay-and-sum beamforming simulations and results of both MATLAB and CUDA. A discussion on the feasibility of a passive sonar system on an AUV is also presented.
- **Chapter 5** outlines the conclusions and recommendations for future work.

Chapter 2
Passive Sonar System Overview and Theory

This chapter will present an overview of a generic passive sonar system and investigate each of its functional units with respect to theory in a sequential manner. Also presented is the passive sonar equation that forms the basis of performance prediction of a passive sonar system (see section 2.7).

2.1 System Elements

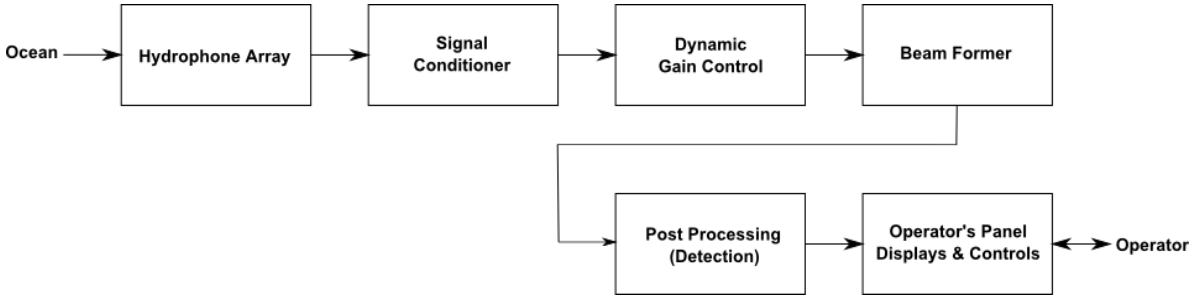

Figure 2.1: Generic passive sonar system [3] [5].

Figure 2.1 shows the functional units of a generic passive sonar system. Acoustic waves propagating in the ocean are received by the hydrophone array (see section 2.2) after which pre-amplification and filtering in the desired frequency band is carried out by the signal conditioner. Dynamic gain control is used to ensure a constant signal level for effective digitisation of the analogue signals (see section 2.3). The digital beamformer's output consists of the number of beams formed at all the positions or in all the directions of interest (see section 2.4) [3]. Post processing involves detection that is a statistical process of deciding whether there is a target of interest present in the received signal or not (see section 2.5). The operator's panel displays and controls can consist of a number of different components and combinations of these components (see section 2.6). Examples of operator displays are Time-Bearing, LOw Frequency and Analysis Recording (LOFAR) and Frequency-Bearing displays. Possible control functions include switching between different beams or setting the detection threshold.

2.2 Hydrophone Array

2.2.1 Directivity of Transducers and Arrays

A transducer is a device that converts energy from one form to another with a known relationship between the input and output energy of the transducer. An electroacoustic transducer converts electrical energy into underwater acoustic energy and vice versa. When an electroacoustic transducer is used only to receive underwater acoustic energy and convert it to electrical energy at its output, it is called a receiver or hydrophone. The reverse of this process is called a transmitter or projector [1]. There exist different types of electroacoustic transducers namely piezoelectric, its counterpart electrostrictive and magnetostrictive [1]. The inner workings of these different types of electroacoustic transducers are outside of the scope of this thesis.

A transducer that receives or transmits signals equally in all directions is called an omnidirectional transducer. On the other hand, directivity is the term used to describe a transducer that receives or transmits signals only in certain directions and is a measure of the transducer's directionality. By combining omnidirectional hydrophones spatially to form an array, this same effect can be achieved. A receiving transducer or array with spatial directivity can be seen as a spatial filter of signals [6]. Noting in "Transducers and Arrays for Underwater Sound", by Sherman, Charles H., Butler, and John L. [7] that the directivity of a receiving or transmitting transducer or array is analogous due to acoustic reciprocity and that the focus of this thesis is on passive sonar, it will be assumed from here onwards that the array in question will be a hydrophone (or receiving) array to which will be referred to only as the "array". An array element (hydrophone) will be referred to as a "sensor".

2.2.1.1 Array Gain

Array gain is a measure of an array's ability to discriminate signals of interest from noise. The directional response (see section 2.2.2) of an array originates from the difference in properties of the received signal and noise. Ambient noise in the ocean arrives at the sensors with a small amount of correlation while signals arriving from point sources will have high correlations when received by the sensors [1] [8]. To illustrate this effect, consider the example of a sinusoidal plane wave signal arriving parallel to a linear array of two sensors. Each sensor will receive the signal with equal amplitude and phase while noise will arrive with random amplitude and phase. With *d* the distance between the two sensors, the output of the array for this signal will be [8]:

$$S_{array} = 2S\sin(\omega t + \varphi) \qquad \textbf{2.2.1}$$

Squaring this value and averaging over time yields the mean square output [8]:

$$Power_{signal} = \frac{\int_0^{2\pi/\omega}\int_0^{2\pi}[2S\sin(\omega t+\varphi)]^2\,d\varphi dt}{\int_0^{2\pi/\omega}\int_0^{2\pi}\,d\varphi dt} = 2S^2 \qquad \textbf{2.2.2}$$

If the same scenario is applied to a single sensor, S_{array} will not be multiplied by a factor of 2 and consequently have an average power of $S^2/2$ which is a factor of four lower than with two sensors.

The average power of the received noise is given by [8]:

$$Power_{noise} = \frac{\int_0^{2\pi/\omega}\int_0^{2\pi}[S\sin(\omega t)+S\sin(\omega t+\varphi)]^2\,d\varphi dt}{\int_0^{2\pi/\omega}\int_0^{2\pi}l\,d\varphi dt} = S^2 \qquad \textbf{2.2.3}$$

Note that the average power of the noise has only doubled which leads to a Signal to Noise Ratio (SNR) improvement by a factor of two. In the ideal case of truly uncorrelated noise and a correlation of one between received signals, the factor of improvement in SNR will be equal to the number of sensors. This improvement of signal-to-noise is referred to as the *array gain* [8].

Array gain is defined as the ratio between the increase in SNR of the array and the SNR of a single array element [3] [7]:

$$AG = \frac{\langle S^2\rangle/\langle N^2\rangle}{\langle s^2\rangle/\langle n^2\rangle} \qquad \textbf{2.2.4}$$

It is commonly expressed in decibels [7]:

$$AG = 10\log\frac{\langle S^2\rangle/\langle N^2\rangle}{\langle s^2\rangle/\langle n^2\rangle} \qquad \textbf{2.2.5}$$

- $\langle S^2\rangle$ and $\langle N^2\rangle$ - Mean square signal and noise outputs of the array respectively
- $\langle s^2\rangle$ and $\langle n^2\rangle$ - Mean square of the signal and noise output of a single element of the array respectively

2.2.1.2 Directivity Amplitude Function

The directivity amplitude function is a representation of the directional response of an array's gain with respect to angle. It is defined as the ratio between the *acoustic pressure amplitude* in a given direction and the *maximum acoustic pressure amplitude* in the principal direction of reception (using polar coordinates) [3]:

$$DAF(\theta,\varphi) = \frac{p(t,\theta,\varphi)}{p(t,\theta_0,\varphi_0)} \qquad \textbf{2.2.6}$$

- $p(t,\theta,\varphi)$ - Pressure amplitude at range r in the direction θ,φ

- $p(t, \theta_0, \varphi_0)$ - Pressure amplitude at range r in the principal direction of reception θ_0, φ_0

2.2.1.3 Directivity Function

The ratio between the *acoustic intensity* in a given direction and the *maximum acoustic intensity* in the principal direction of receiving is the directivity function of an array [3]:

$$\Gamma^2(\theta, \varphi) = \frac{I(\theta, \varphi)}{I(\theta_0, \varphi_0)} \qquad \textbf{2.2.7}$$

- $I(\theta, \varphi)$ - Intensity at range r in the direction θ, φ
- $I(\theta_0, \varphi_0)$ - Intensity at range r in the principal direction of reception θ_0, φ_0

It is commonly expressed in decibels:

$$\begin{aligned} B(\theta, \varphi) &= 10 \log \Gamma^2(\theta, \varphi) \\ &= 20 \log \Gamma(\theta, \varphi) \end{aligned} \qquad \textbf{2.2.8}$$

2.2.1.4 Directivity Factor and Directivity Index

The directivity factor and directivity index of an array are measures of an array's ability to discriminate between plane waves received in the principal direction of reception and plane waves incident on the array from other directions [7]. In the special case where the incident signal on the array is a coherent plane wave and the received noise is isotropic and incoherent, the array gain reduces to the directivity factor. The directivity factor of an array is defined by the following relationship [9]:

$$D_f = \frac{peak\ intensity\ of\ radiated\ pattern}{average\ intensity\ of\ radiated\ pattern} \qquad \textbf{2.2.9}$$

The directivity index is defined as [7]:

$$DI = 10 \log D_f \qquad \textbf{2.2.10}$$

2.2.2 Beam Pattern

The three-dimensional plot of an array's directivity function is commonly referred to as the directivity pattern or beam pattern of the array [3]. A typical example of an array's beam pattern (in polar coordinates) is plotted in Figure 2.2 below. Note that this plot is two-dimensional because of the fact that the beam pattern of the array is rotationally symmetrical and thus there is no loss of information.

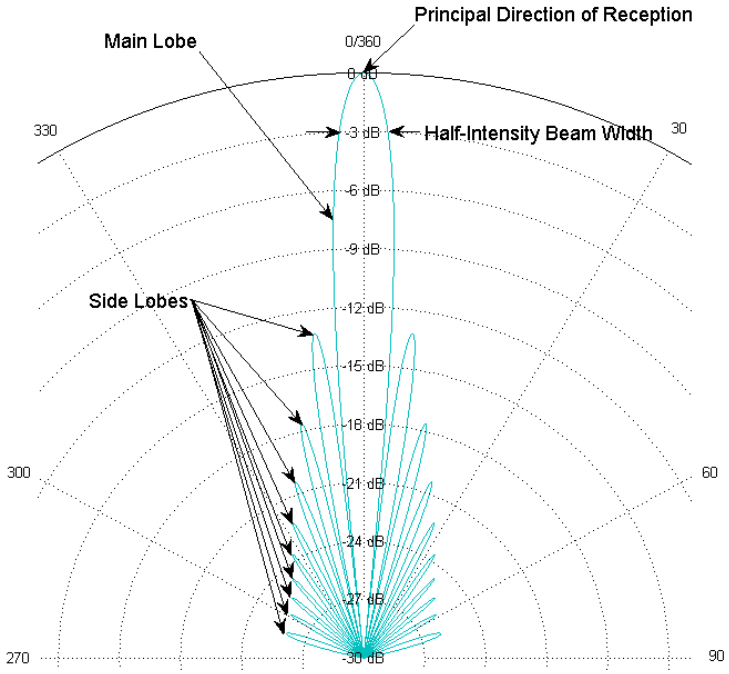

Figure 2.2: Plot of a directivity function in polar coordinates (section through z-axis)

Important things to note in Figure 2.2 are the main lobe, side lobes, half-intensity beam width and the principal direction of reception.

Main lobe: This lobe represents the array's principal direction of reception [3]. The point at which this lobe peaks can be thought of as the direction in which the array is "looking".

Side lobes: These lobes represent other directions at which the array receives signals [3]. These are unwanted, as they will add signals from directions other than the principal direction of reception, consequently reducing the SNR of the array and making it more difficult for the array to distinguish the direction from where the signal of interest actually came from.

Half-intensity beam width: This angle represents the main lobe width at half of its acoustic intensity or -3 dB gain [3].

It was mentioned in section 2.2.1 that the directional response of an array arises from the difference in properties between the received signal and noise. To investigate this effect with respect to beam pattern, consider an unshaded (all sensor outputs are summed equally, refer to section 2.2.3 for an explanation of shading an array) linear array with $\lambda/2$ spacing between

array elements. The directivity amplitude function of an unshaded linear array of N elements with spacing d is defined by the equation [3] [8]:

$$DAF(\theta) = \frac{\sin\left(N\pi d\frac{\sin\theta}{\lambda}\right)}{N\sin\left(\pi d\frac{\sin\theta}{\lambda}\right)} \qquad \textbf{2.2.11}$$

To get the directivity function or beam pattern of the array, the directivity amplitude function must be squared to get the power response and expressed in decibels [8]:

$$\begin{aligned} B(\theta) &= 10\log\{[DAF(\theta)]^2\} \\ &= 10\log\left\{\left[\frac{\sin\left(N\pi d\frac{\sin\theta}{\lambda}\right)}{N\sin\left(\pi d\frac{\sin\theta}{\lambda}\right)}\right]^2\right\} \end{aligned} \qquad \textbf{2.2.12}$$

By varying the number of sensors (N) in the array and plotting the beam pattern for each value in MATLAB with $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$, it becomes clear in Figure 2.3 that the directional response and thus the SNR in the principal direction of reception improves as the number of sensors increases, caused by the array receiving less noise from other directions than the principal direction of reception.

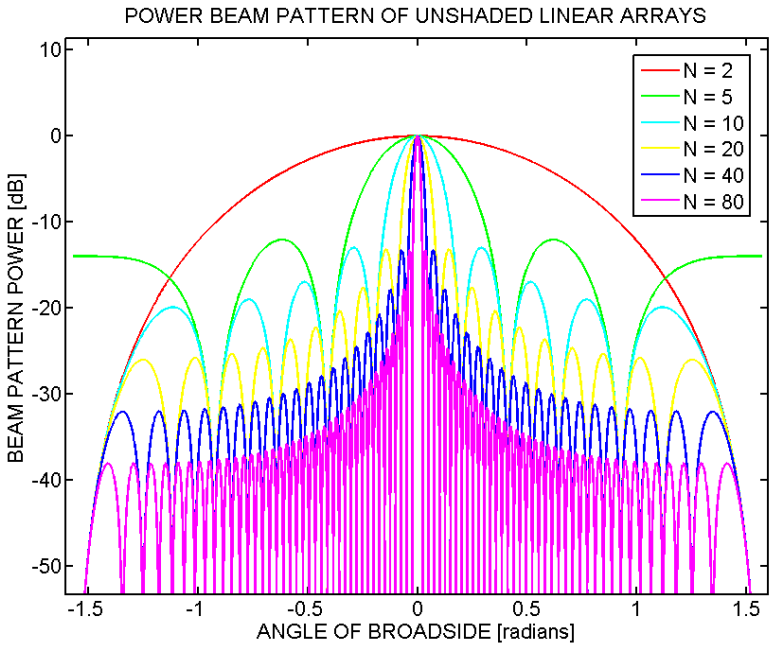

Figure 2.3: Power beam patterns of unshaded linear arrays with a variable number of elements.

2.2.3 Shading Functions

Shading is a method used for achieving partial control over the beam pattern of an array having a particular geometry. Amplitude shading is commonly used to adjust the amplitude response of the array elements. Usually its purpose is to adjust the beam pattern of the array to achieve maximum response in the centre and a minimum response at its extremities, thus maximizing the main lobe response and minimizing the side lobes' responses [3] [1]. Phase shading is also used in arrays and this kind of shading entails varying the spacing of elements in the array to achieve a desired beam pattern. It is commonly used in radio antennae design but not in underwater sound applications [1].

In acoustic sonar array design, there are three main objectives, the **first** is to have the narrowest possible main lobe, the **second** being that this lobe is as sensitive as possible at its peak and the **third** is to decrease side lobe levels to a minimum [10]. There exists a relationship between the first and third objectives that causes a trade-off between main lobe width and side lobe levels. By decreasing the side lobe levels via amplitude shading the main lobe width will increase and thus will also decrease the angular resolution in the principal receiving direction and vice versa [10].

There are many types of shading functions for example rectangular (which is the equivalent of no shading), triangular, binomial, hanning, its counterpart hamming, Gaussian, and Dolph-Chebyshev [3] [11]. As mentioned above, it is favourable to have the narrowest possible main lobe to achieve good resolution and to have minimal side lobe levels to achieve maximum SNR. The shading function which achieves this objective optimally is the Dolph-Chebyshev shading function as it achieves the narrowest main lobe width for a given side lobe level [1].

In Figure 2.4 (Gaussian shading), Figure 2.5 (Hamming shading) and Figure 2.6 (Dolph-Chebyshev shading) the beam patterns of a 20 element array with various shading functions was plotted in MATLAB. Notice the decreased side lobe levels, increased main lobe width, and decreased sensitivity at the main lobe peak when shading is applied. Figure 2.7 shows the Dolph-Chebyshev shaded beam pattern of an array with various specifications for the attenuation of the side lobe levels. Notice how the main lobe width increases as the side lobe level attenuation increases.

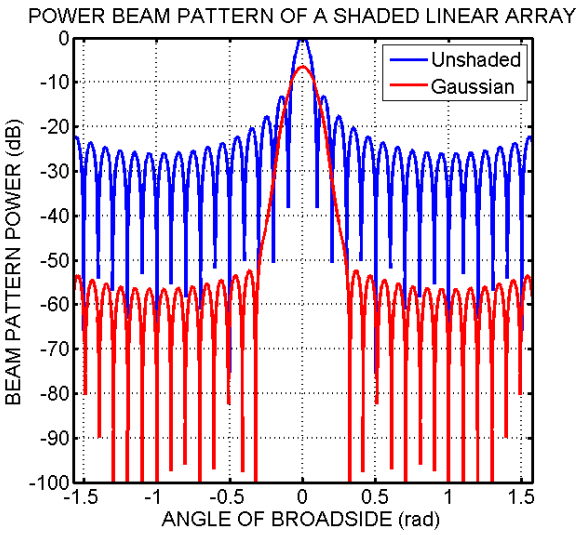

Figure 2.4: Power beam pattern of a Gaussian shaded, 20-element array.

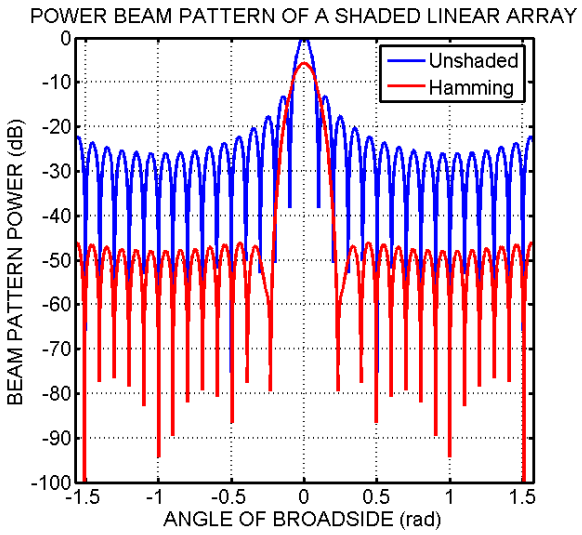

Figure 2.5: Power beam pattern of a Hamming shaded, 20-element array.

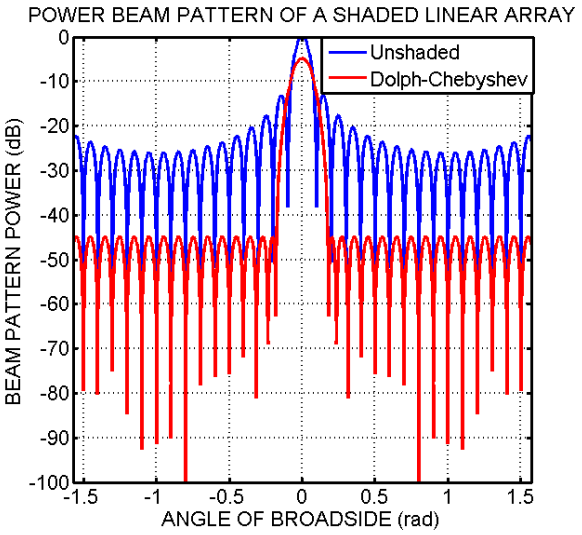

Figure 2.6: Power beam pattern of a Dolph-Chebyshev shaded (40dB attenuation of side lobes), 20-element array.

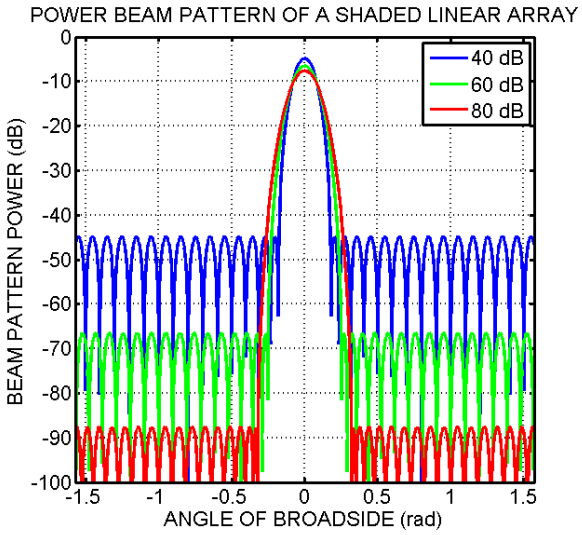

Figure 2.7: Power beam pattern of a Dolph-Chebyshev shaded, 20-element array with various specifications for the side lobes' attenuation.

2.2.4 Beam Steering

Beam steering is the process of changing the principal direction of reception of an array [3]. In the example of section 2.2.2 it was assumed that the planar sinusoidal signal arrives parallel to the linear array and thus each sensor in the array received the signal equally and in phase. When the signal arrives from a different angle, each of the sensors will receive the signal at a different point in time and therefore out of phase with respect to the first sensor that received the signal (see Figure 2.8). If the array is not steered in the direction of the signal, the signals from the sensors will be summed out of phase, negating the constructive interference of the signals and hence degrading the array gain.

To compensate for this, the array can be steered mechanically or electronically by introducing a time delay between sensors so that the principal direction of reception of the array is in the direction from where the signal came from. Electronic steering of an array has an advantage over mechanical steering because the beams in all directions can be formed simultaneously by using pre-formed beams or in other words pre-calculated time delays between sensors [3]. Broadband receiving systems such as passive sonar systems use time delay beamforming [3].

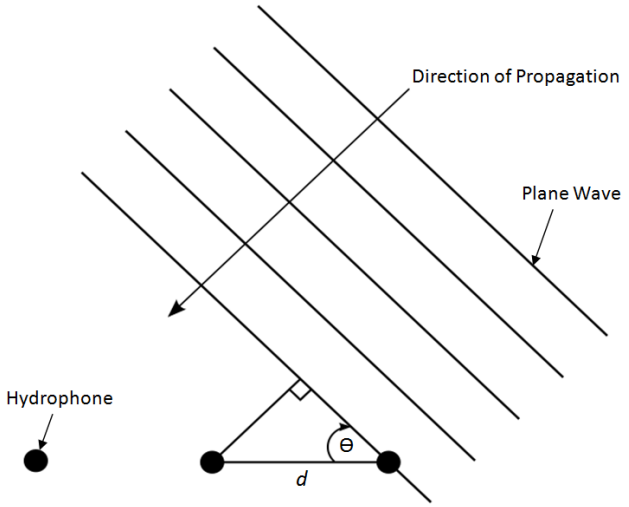

Figure 2.8: Plane wave incident on a linear array with uniform spacing between sensors.

Referring to Figure 2.8, the signal arriving at each sensor is given by [8]:

$$S_j = S\cos\left[\omega\left(t + \tau_j\right)\right] \qquad \textbf{2.2.13}$$

where the time difference for the *j*'th sensor is [8]:

$$\tau_j = \frac{jd}{c}\sin\theta \qquad \textbf{2.2.14}$$

and the phase difference for the *j*'th sensor is [8]:

$$\varphi_j = \frac{j2\pi d}{\lambda}\sin\theta \qquad \textbf{2.2.15}$$

Incorporating this time delay (or fixed phase shift in the case of a sinusoidal planar wave) into the directivity function (equation 2.2.12) of an unshaded linear array yields [8]:

$$B(\theta) = 10\log\left\{\left[\frac{\sin\left(N\pi d\frac{(\sin\theta-\sin\varphi)}{\lambda}\right)}{N\sin\left(\pi d\frac{(\sin\theta-\sin\varphi)}{\lambda}\right)}\right]^2\right\} \qquad \textbf{2.2.16}$$

Plotting this directivity function in MATLAB for $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$, with $\varphi = 0°, 30°, 45°, 60°, 90°$ and with twenty sensors in the array yields Figure 2.9. Note that the beam width for a linear array increases as the beam is steered towards its endfire directions. Steering a linear array towards its endfire directions decreases the effective aperture area of the array [3]. Effective aperture area can be thought of as the "capture area" that the array has to receive power from the plane wave. Decreasing it leads to less received power from the plane wave incident on the array, causing an increase in main lobe width [12]. In Figure 2.10 it is apparent that when the array is steered towards the plane wave in the top of the figure, causing a steering angle closer to the array's endfire direction when compared to the bottom plane wave, the effective aperture of the array decreases.

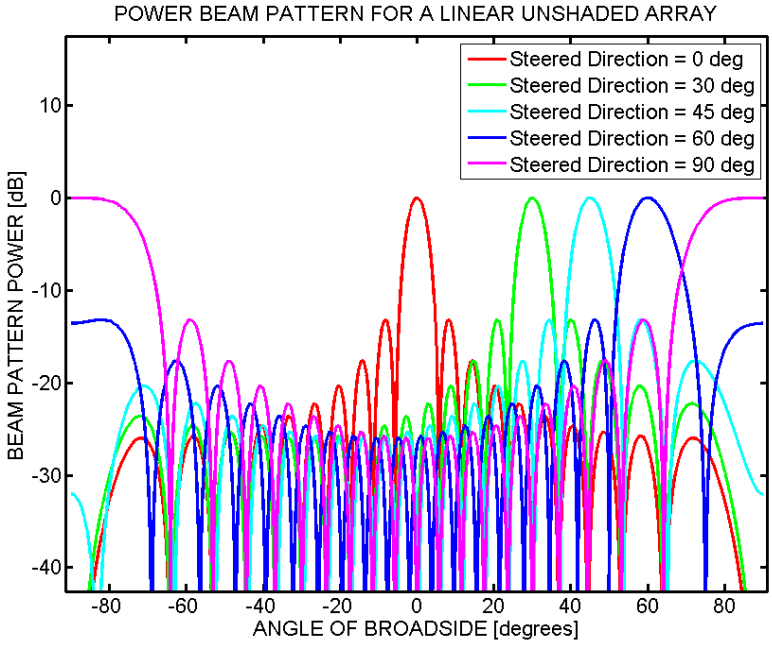

Figure 2.9: Power beam pattern for a linear unshaded array steered to multiple directions.

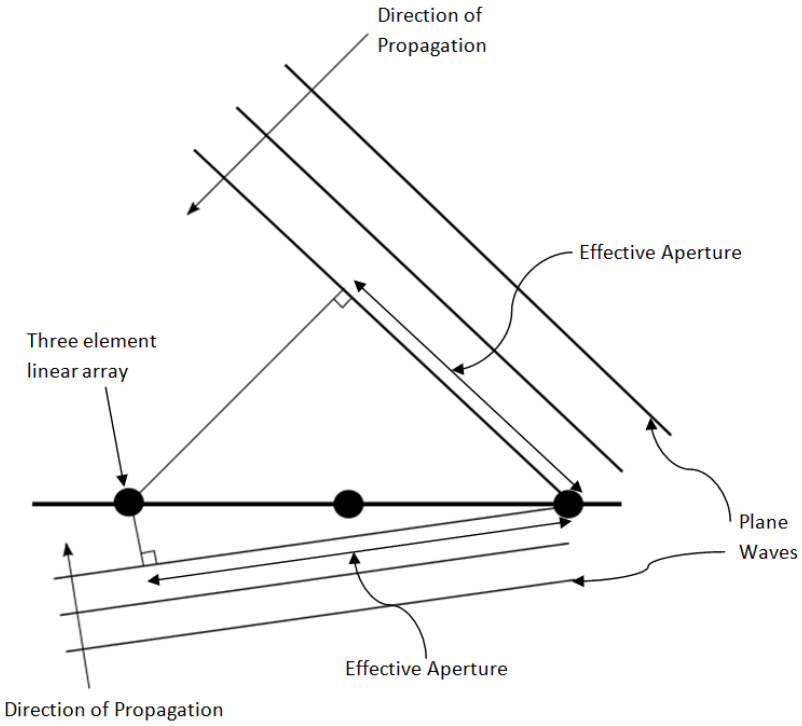

Figure 2.10: Two plane waves incident on a linear array.

2.2.5 Sensor Spacing

The uniform or non-uniform spacing between sensors plays a crucial role in the beam pattern of arrays. For a linear array, λ/2 spacing between elements is referred to as the *design frequency* [8]. The reason why it is called the design frequency, where in fact it is only the geometric spacing between sensors, can be explained with the help of the Nyquist-Shannon sampling theorem. The Nyquist-Shannon sampling theorem states that a signal must be digitally sampled at a rate higher than the Nyquist rate, which is given by [13]:

$$f_{s\,(Nyquist)} = 2B \qquad \textbf{2.2.17}$$

- B - Bandwidth of a bandlimited signal
- $f_{s\,(Nyquist)}$ - Nyquist sampling frequency

From equation 2.2.17 it can be concluded that there must be a minimum of two samples taken for every period of the signal. When a signal is sampled at a frequency lower than the Nyquist frequency, called undersampling, aliasing occurs. When a signal is undersampled, perfect signal reconstruction from digital samples is not possible and the reconstructed signal does not represent the initial pre-sampled analogue signal. Aliasing can be clearly seen in the digital frequency spectrum of a signal. When aliasing occurs, each period of the digital frequency spectrum of the signal starts to overlap with the next (or previous) period and thus distorts the frequency characteristics of the signal (see Figure 2.11 below).

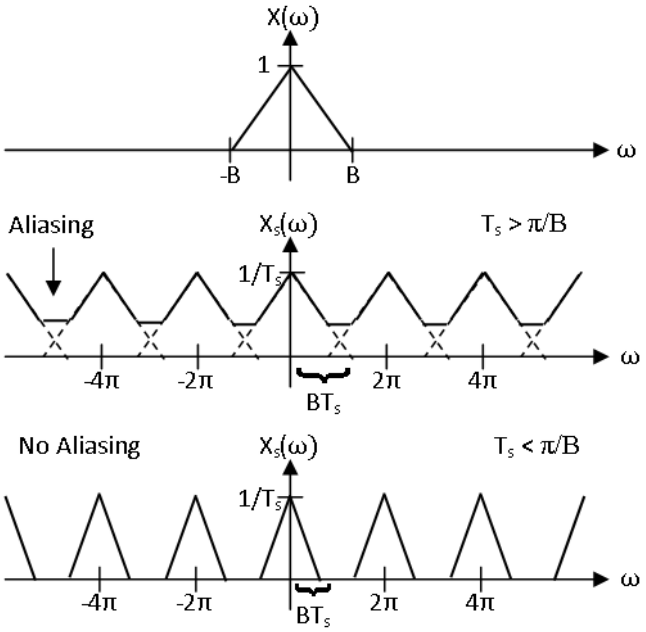

Figure 2.11: Frequency spectrum of a signal $X(\omega)$ and its aliased and non-aliased sampled spectrums $X_s(\omega)$ [14]

To explain why a spacing of λ/2 is considered the design frequency of an array, an example of *spatial* aliasing in the field of acoustic imaging presented by Scholte et al. [15] will be considered.

In the acoustic imaging practice, two-dimensional area measurements are made of acoustic waves with the use of a planar array situated a certain distance from the acoustic source. Sampling a signal in space is called spatial sampling and can be compared to sampling a continuous time signal in time where instead of continuous time, continuous position is sampled.

A signal propagating in continuous time has an angular frequency ($\omega=2\pi f$) in radians per second whereas a signal propagating in space and time (spatiotemporal signal) has a spatial frequency (or *wavenumber*) as well and is given by [12]:

$$k = \frac{\omega}{v} = \frac{2\pi f}{v} = \frac{2\pi}{\lambda}\ rad/m \qquad \textbf{2.2.18}$$

- λ - Wavelength
- f - Frequency
- v - Linear speed

Taking the Fourier transform of a spatiotemporal signal with respect to wavenumber and temporal frequency yields the signal's wavenumber-frequency spectrum [6]. Taking the Fourier transform with respect to wavenumber only yields the wavenumber spectrum. A typical wavenumber spectrum of a spatial domain signal contains all the spatial frequencies and their specific energies that make up the spatial waveform (see Figure 2.12). To avoid spatial aliasing,

it is necessary to sample at a rate of twice the highest wavenumber or in other words at the Nyquist sampling wavenumber and is given by:

$$\begin{aligned} k_{Nyquist} &= 2k_{max} \\ &= 2\left(\frac{2\pi}{\lambda_{min}}\right) rad/m \end{aligned} \qquad \textbf{2.2.19}$$

To get the spacing needed to avoid spatial aliasing:

$$\begin{aligned} \frac{2\pi}{\lambda_{Nyquist}} &= \frac{4\pi}{\lambda_{min}}\ rad/m \\ \frac{1}{\lambda_{Nyquist}} &= \frac{2}{\lambda_{min}}\ m^{-1} \\ \lambda_{Nyquist} &= \frac{\lambda_{min}}{2}\ m \end{aligned} \qquad \textbf{2.2.20}$$

From equation 2.2.20, $\lambda_{Nyquist}$ can be thought of as the spatial wavelength of the array needed to avoid spatial aliasing and is the reason why λ/2 is can be seen as the design frequency of the array.

To illustrate this concept, consider two waveforms with different frequencies and directions of propagation in Figure 2.12 with their respective wavenumber spectrum next to them. The dark areas of the waveforms represent high sound pressure whereas light areas represent low sound pressure. It is clear from the waveform and its wavenumber spectrum that waveform (a) has a higher frequency than waveform (c).

By combining the two waveforms of Figure 2.12 results in the waveform and wavenumber spectrum shown in Figure 2.13. Note that the wavenumber spectrum of each wave is clearly distinguishable from the other in Figure 2.13 (b).

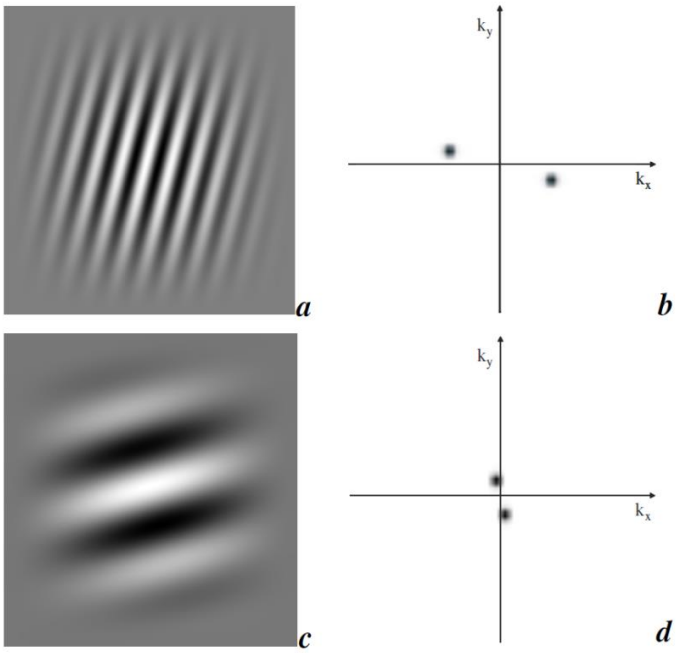

Figure 2.12: Two separate waveforms (a and c) with their respective wavenumber spectrum to the right (b and d) [15].

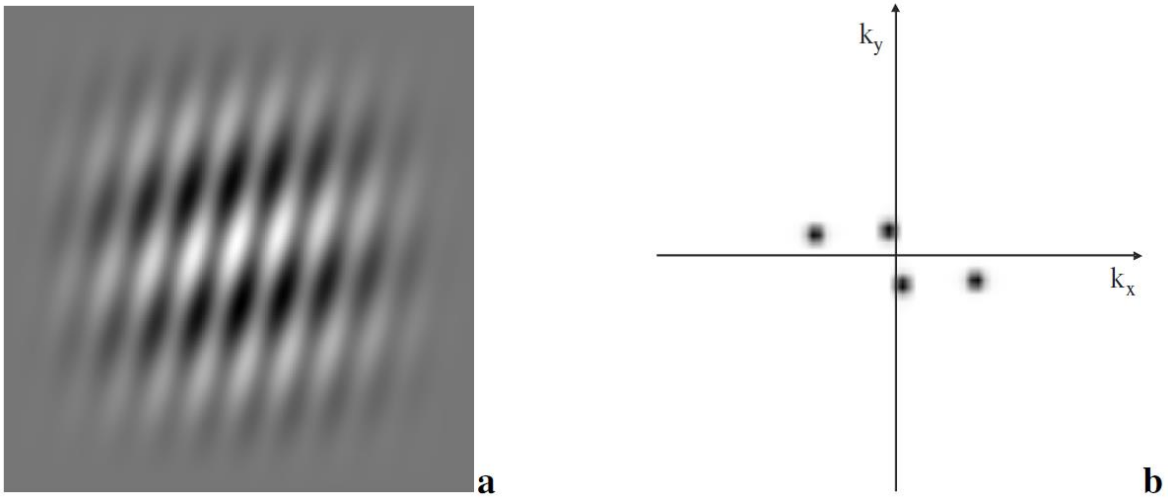

Figure 2.13: Two waveforms from Figure 2.12 combined (a) with its wavenumber spectrum (b) [15].

By sampling the waveform of Figure 2.13, but at a rate lower than twice the highest wavenumber, the wavenumber spectrum becomes aliased as shown in Figure 2.14. In this wavenumber spectrum, the area highlighted in grey shows the original wavenumber spectrum of the wave but just as undersampling in the time domain causes an overlap of the periodical frequency spectrum, undersampling in the spatial domain causes an overlap of the periodical wavenumber spectrum of the wave as well and is present within the dashed line square of Figure 2.14. Taking the inverse transform of the spectrum in Figure 2.14 results in a waveform that is not a representation of the original waveform and can be seen in Figure 2.15 below.

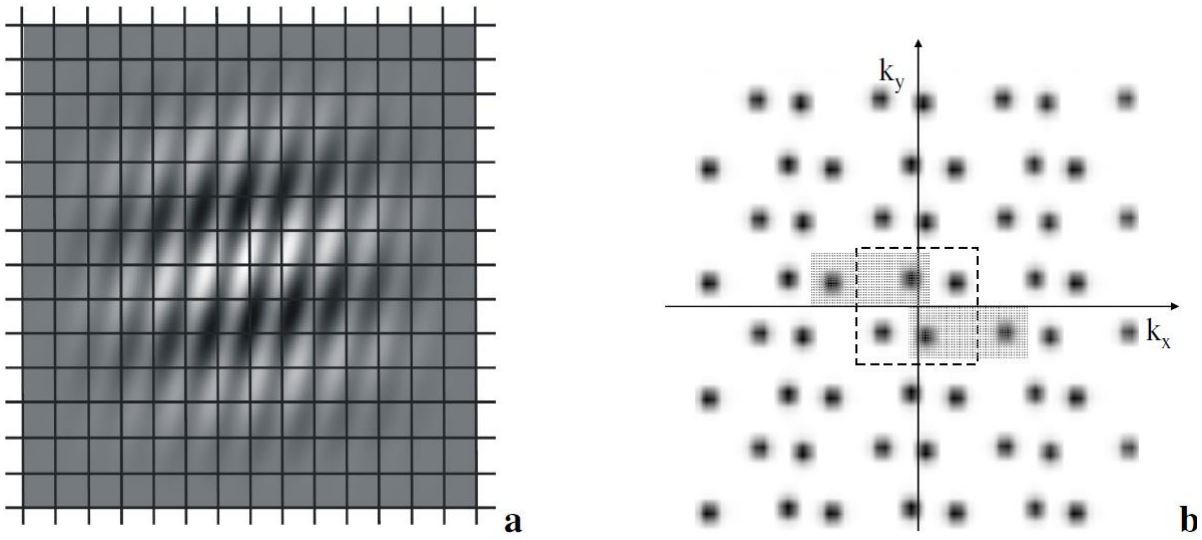

Figure 2.14: Undersampled version of the spatial wave in Figure 2.13.

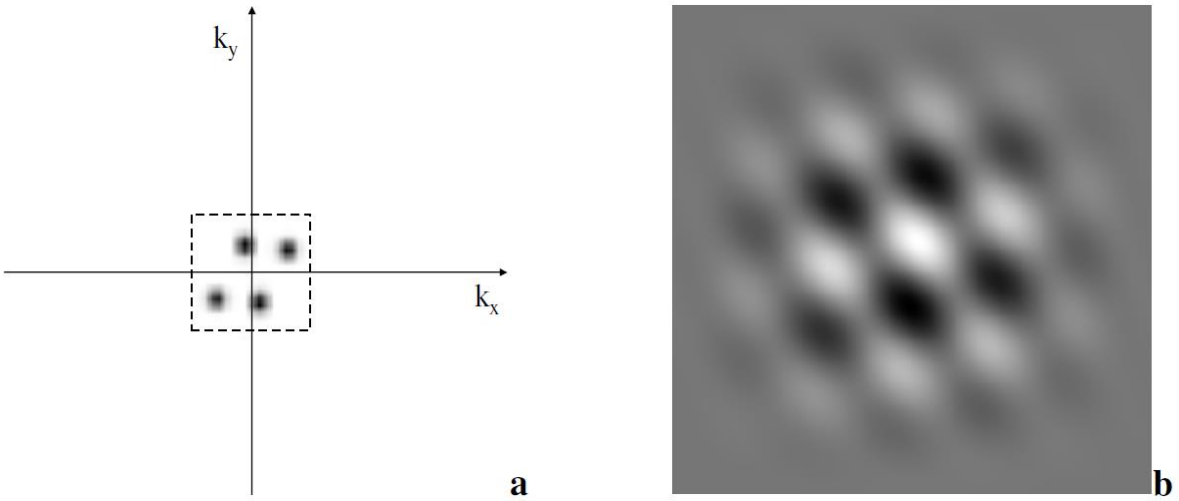

Figure 2.15: Inverse transform of the wavenumber spectrum in Figure 2.14.

The effects of spatial aliasing can also be observed in the beam pattern of a linear array. Here, spatial undersampling will result in side lobes close- or equal to the main lobe magnitude. These huge side lobes are called *grating lobes* and degrade the performance of the array greatly due to consequent ambiguities in calculation of the direction of arrival of the wave incident on the array.

In Figure 2.16, the beam pattern of an unshaded linear array with 20 sensors was plotted in MATLAB with various lengths of spacing between the sensors to illustrate this effect. Notice the grating lobe that appears at 90˚ when the sensor spacing is equal to λ. This grating lobe causes the array to be unable to distinguish between a plane wave incident at 0˚ or 90˚.

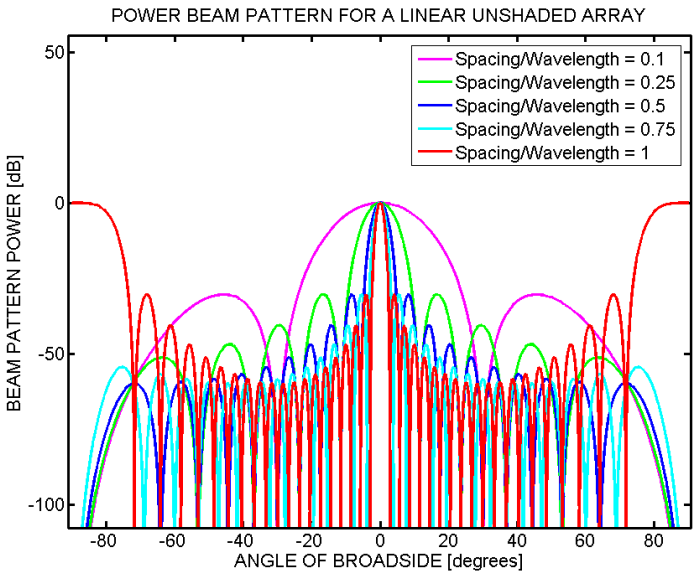

Figure 2.16: Power beam pattern for a linear unshaded array with different lengths of spacing between sensors

2.2.6 Array Ambiguity

Assuming the array beam pattern has no unusual grating lobes, linear arrays suffer from direction of arrival ambiguity in two- and three-dimensional space.

In section 2.2.4, the time delays between sensors of a linear array is given by equation 2.2.14. This equation is independent of which side the signal is incident on the array and hence causes the array to suffer from left-right ambiguity in two-dimensional space [8]. To illustrate this effect, note in Figure 2.17 that the arrival time of both plane waves are equal at each sensor and hence indistinguishable for the linear array.

This ambiguity also exhibits itself in the beam pattern of the array and can be seen in Figure 2.18 where a copy of the main lobe exists at 180˚.

Intuitively, a one-dimensional array suffers from left-right ambiguity in two-dimensional space and in three-dimensional space it suffers from depression-elevation (or up-down) ambiguity as well.

Applying the same reasoning for a two-dimensional (plane) array, in two-dimensional space it will not suffer from any ambiguities whereas in three-dimensional space it will suffer from depression-elevation ambiguity.

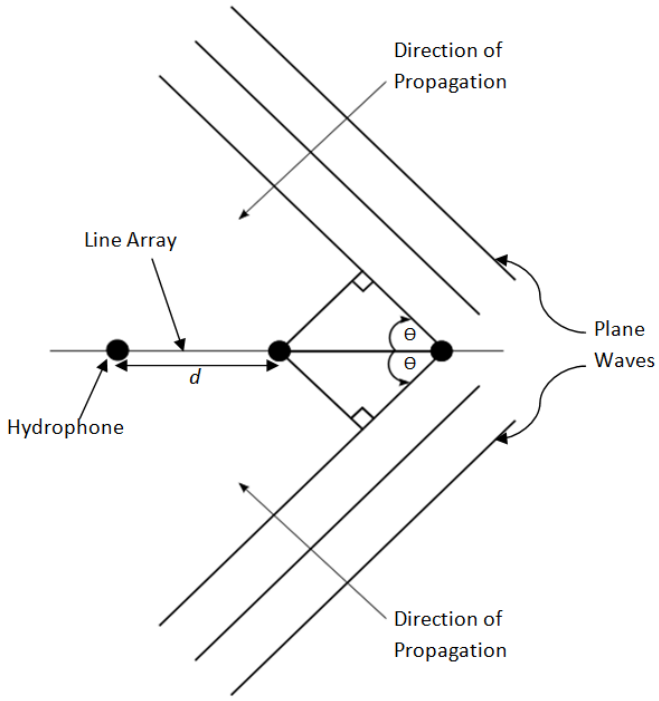

Figure 2.17: Plane waves incident on a linear array with uniform spacing between sensors.

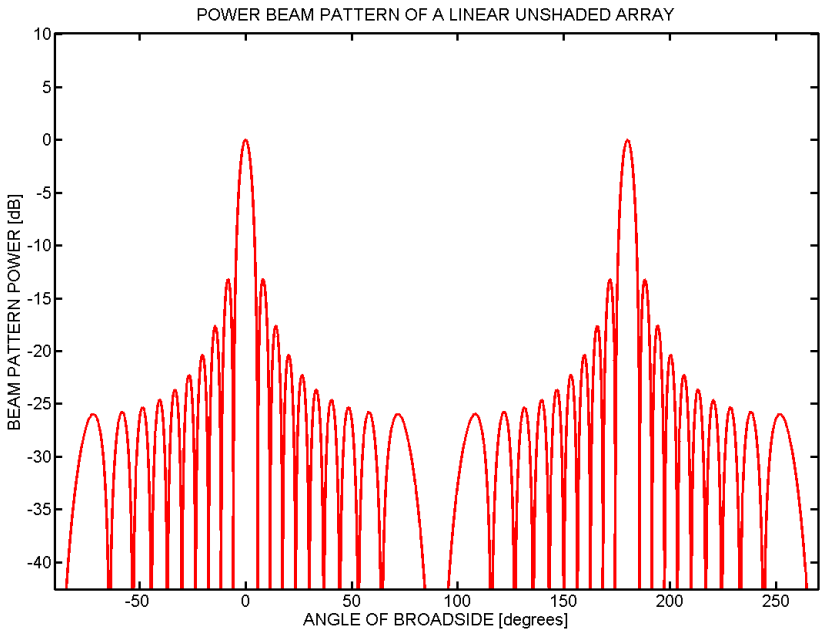

Figure 2.18: Power beam pattern of a linear unshaded array.

2.2.7 Near-Field and Far-Field

In acoustic array theory, it is often assumed that the distance between the source and the sensor are large enough so that the incident wave is perceived as planar by the array [16]. However,

when the receiving array is close enough to the source, this assumption is invalid and the wave incident on the array is perceived as circular. A simple illustration in Figure 2.19 demonstrates this effect. Notice that the angle Θ becomes smaller as the array moves further from the source. When the array moves far enough from the source, the angle Θ becomes zero and the array perceives the circular wave as planar.

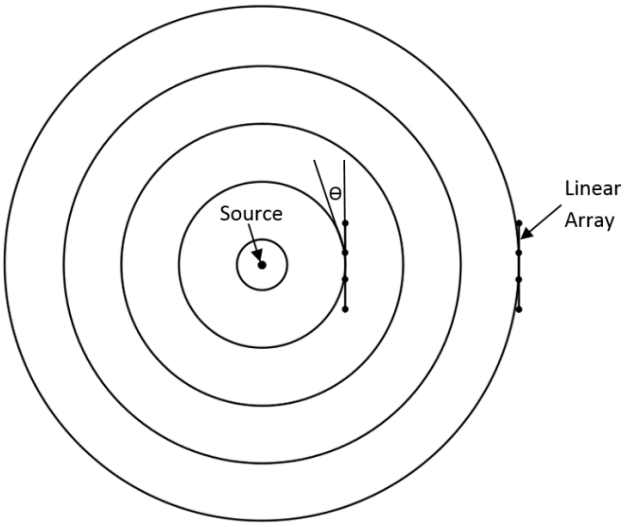

Figure 2.19: Linear array in the vicinity of a source with circular waves.

Beamforming in the near-field can still be performed but with different delays applied to each sensor to compensate for the difference of arrival of the circular wave (corresponding to the shortest distance from the source to a sensor in the array) between the sensors of the array [16]. Unlike far-field beamforming where only direction of arrival (bearing) can be calculated, near-field beamforming has the advantage of being able to calculate the range of the source and hence the exact position of it [6]. An approximation of the distance at which far-field beamforming can be applied is commonly defined as a multiple of the following equation [16] [3]:

$$r = \frac{L^2}{\lambda} \qquad \textbf{2.2.21}$$

- r - Distance from the source to the array
- L - Maximum dimension of the array
- λ - Operating wavelength of the array

Ryan [16] states that 2.2.21 is an oversimplification of the near-field distance because it does not take into account the angle of incidence of the wavefront on the array. Consider the response $G(r, \theta)$ of a uniformly weighted (no shading) near-field delay-and-sum beamforming array in Figure 2.20 to a monochromatic spherical wave [16]:

$$G(r,\theta) = \sum_{m=0}^{M-1} \frac{e^{-j(kr_m+\omega\tau_m)}}{r_m} \qquad \textbf{2.2.22}$$

- k - Wavenumber
- ω - Angular frequency
- r_m - Distance from the source to the *m'th* array sensor
- τ_m - Beamformer delay for the *m'th* sensor

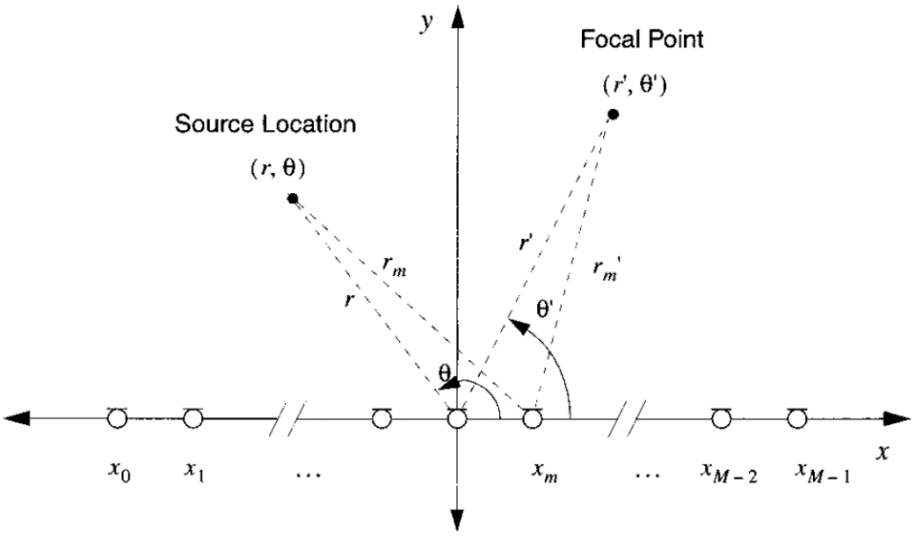

Figure 2.20: Linear array with a source located in the near-field [16].

The equation for r_m and τ_m for a near-field source from Figure 2.20 [16]:

$$r_m = \sqrt{r^2 - 2rx_m\cos\theta + x_m^2} \qquad \textbf{2.2.23}$$

$$\tau_m = \frac{r' - r_m'}{c} \qquad \textbf{2.2.24}$$

where c is the speed of wave propagation. The beamformer delays (τ_m) are used to steer the beam of the array to a point of interest (r', θ') in Figure 2.20. When the focal point of the array matches the propagation delays of the source such that $(r,\theta) = (r',\theta')$, equation 2.2.22 reduces to [16]:

$$G_s(r,\theta) = e^{-jkr} \sum_{m=0}^{M-1} \frac{1}{r_m} \qquad \textbf{2.2.25}$$

For plane wave beamforming, the delays only depend on angle of incidence [16]:

$$\tau_m = \frac{x_m \cos\theta'}{c} \qquad \textbf{2.2.26}$$

Steering the array such that $\theta' = \theta$ yields the near-field response for planar wave beamforming [16]:

$$G_p(r,\theta) = \sum_{m=0}^{M-1} \frac{e^{-jk(r_m + x_m \cos\theta)}}{r_m} \qquad \textbf{2.2.27}$$

As expected, the equation above shows that the beamformer delays do not match the spherical wave's propagation delays between sensors because of the fact that planar wave beamforming is applied with increasing error in delays as the source moves closer. The error produced by this misassumption can be expressed as [16]:

$$\epsilon(r,\theta) = 20\log_{10}\left|\frac{G_p(r,\theta)}{G_s(r,\theta)}\right| \qquad \textbf{2.2.28}$$

Plotting this error response with rectangular coordinates yields Figure 2.21. This figure shows the error response being dependant on distance and angle rather than only distance as stated in equation 2.2.21.

Ryan [16] proposes a new estimate equation for the minimum distance at which plane wave beamforming can be applied for a linear array that also makes provision for the steering angle of the beamformer and is given by:

$$r_n = \frac{(L\sin\theta)^2}{2\lambda} + \frac{L}{2}|\cos\theta| - \frac{\lambda}{8} \qquad \textbf{2.2.29}$$

- L - Array length
- Θ - Steering angle
- λ - Wavelength

This equation is derived from minimising the error response instead of defining an allowable error, which is normally the design constraint [16].

When $\Theta = 90^{\circ}$, which is the broadside direction (no steering) of a linear array, equation 2.2.29 reduces to $L^2/2\lambda - \lambda/8$ which is close to the approximation of equation 2.2.21 depending on what the allowable error is defined as (with 1dB allowable error equation 2.2.21 becomes $L^2/2\lambda$) [16]. However, when the steering angle deviates from π/2 it is clear from Figure 2.22, where equation 2.2.29 has been superimposed, that equation 2.2.21 is not sufficient in defining the near-field region of the array [16].

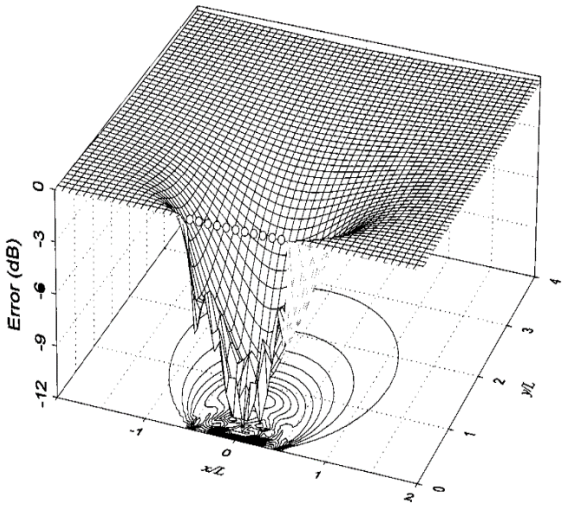

Figure 2.21: Error response of equation 2.2.29 plotted in rectangular coordinates.

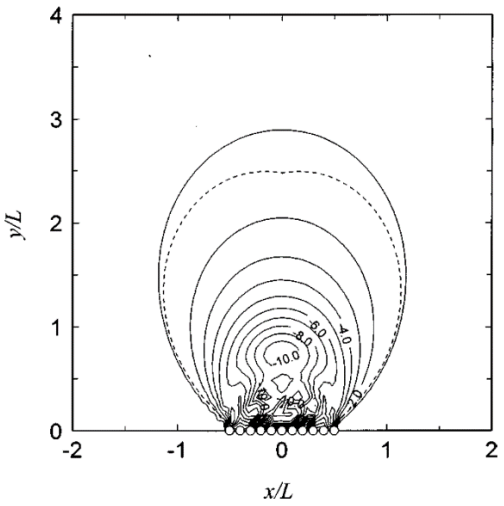

Figure 2.22: Top view of error response in figure to the left with the extent of the calculated near-field distance (equation 2.2.29) superimposed on it with dashed lines.

2.2.8 Array Configuration

The geometric configuration of an array is governed by its performance demands in its specific application or purpose. There exists a wide variety of array configurations and it is outside the scope of this thesis to understand the design, implications and performance of each of these with respect to its application. An overview is presented here.

To name a few commonly encountered array configurations in texts: One-dimensional linear arrays; two-dimensional rectangular, circular and triplet planar arrays; and three-dimensional cylindrical, cubic and spherical volumetric arrays of which some of these may have a uniform or non-uniform spacing between elements [1] [8] [6] [17].

Array configurations that are commonly used in practice include rectangular planar, cylindrical, spherical, conformal and towed linear arrays [7] [17].

Rectangular planar arrays occupy two dimensions as the name indicates. Figure 2.23 shows a large rectangular planar array made up of smaller rectangular planar arrays and is used for deep-water applications [7]. The sensors of these arrays are mounted on an absorbent or reflective baffle (with the distance between the reflective baffle and the sensor small enough to insure minimum phase difference) or on a structure that is acoustically transparent to insure minimal interaction between the wave incident on the array and the mechanical support that the array is mounted on [17]. Transparent structures have the same impedance as the water (or medium) in which they are used [17]. This allows a wave to travel through it with minimal reflection of the wave or bending of the wave's direction of propagation.

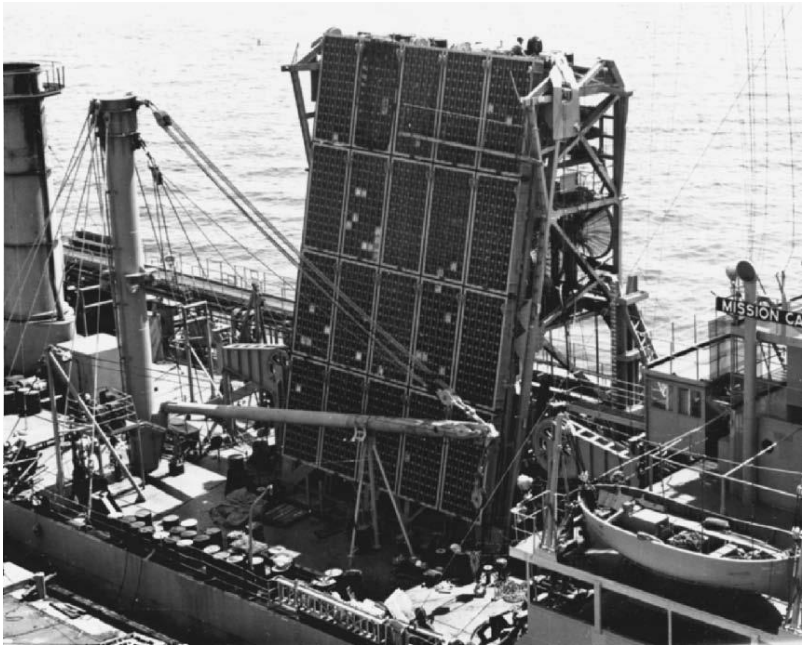

Figure 2.23: Large rectangular planar array made up of smaller rectangular planar arrays [7]

Cylindrical arrays such as the one in Figure 2.24 occupy three dimensions. Three types of construction are used for cylindrical arrays [17]. The first being a cylindrical cast of acoustically transparent material with sleeves introduced during casting where the sensors are inserted. Another construction method uses a cylindrical steel drum with absorbent or reflective baffles mounted on it between each sensor and the steel drum. Lastly, a cylindrical steel drum is used to support vertical columns (Figure 2.24) made of absorbent or reflective material where the sensors are inserted. Each of these constructions use water filled cylindrical steel drums to support the array so that the drums do not succumb to pressure.

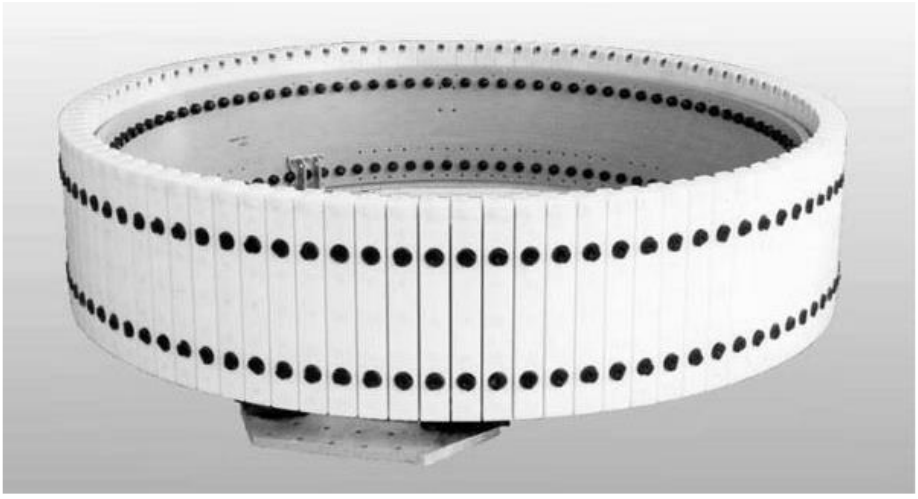

Figure 2.24: Passive sonar cylindrical array used on submarines [18]

Spherical arrays (see Figure 2.25) has an advantage over other array geometries in that they can form equal beams in all directions due to the symmetry of the array [17]. Again, a water filled mechanical support is used on the inside of the array with reflective or absorbent material between the sensor and the mechanical support.

Conformal arrays such as the one in Figure 2.26 takes on the shape of the support structure rather than the other way around and is commonly used on submarines [17]. The form of the support structure and array can also be interdependent on each other. An example of conformal arrays is flank-distributed arrays that submarines use to scan the side (flank) of the vessel. A few conformal arrays such as in Figure 2.26 are mounted on the side of the submarine as depicted in Figure 2.27. These are used to calculate the distance to the source of the acoustic wave incident on the array [17]. The large area covered by the array allows this as it can detect the incident wave's curvature that would otherwise seem like a planar wave to a smaller array.

An interesting type of retractable conformal array for an AUV is shown in Figure 2.28. When retracted, this array takes on the shape of the AUV to minimise water draft over the array while navigating which will otherwise be problematic for a small vehicle like an AUV. More importantly, the small body of an AUV limits the beamforming capabilities of an array that is statically mounted on the body because of the spacing needed between sensors (as discussed in section 2.2.5). The AUV lands on the seafloor and maintains a stationary position where it then deploys the array and collects data for beamforming. When the data is collected, the array retracts and the AUV can move freely with no unnecessary draft caused by the array. When deployed this array has a maximum dimension of 2m and is capable of meaningful beamforming of broadband sources [19].

Figure 2.25: Spherical array for a submarine [7].

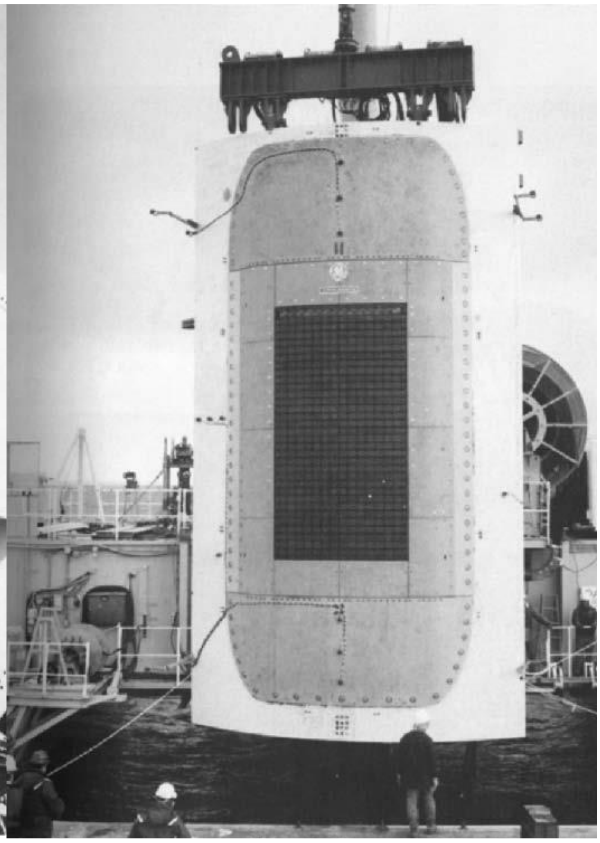

Figure 2.26: A panel of a submarine with a conformal array mounted on it [7].

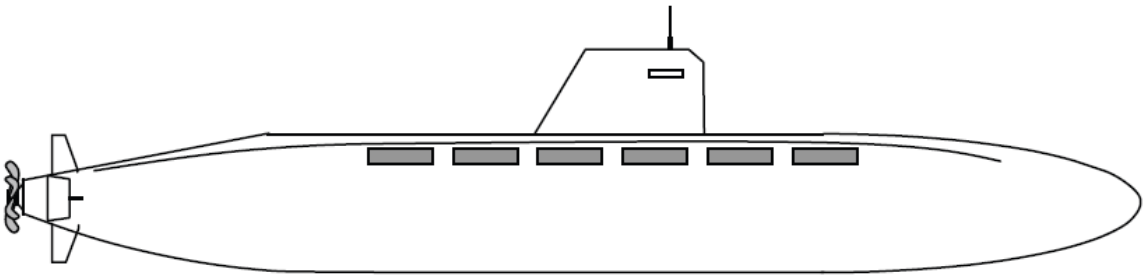

Figure 2.27: Rectangular conformal arrays mounted on the side of a submarine [17].

Figure 2.28: Conformal passive sonar array mounted on an AUV [19].

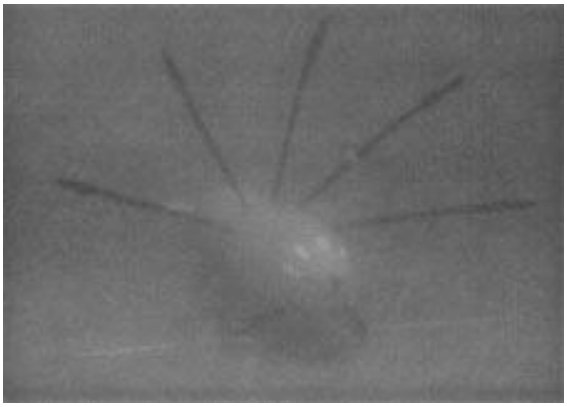

Figure 2.29: Deployed conformal passive sonar array of figure to the left [19].

Towed linear arrays are used to achieve satisfactory directivity functions at low operating frequencies that are otherwise unrealizable with the restriction of the length of a seafaring vessel [17]. Basic construction details of these arrays are depicted in Figure 2.30 where the indicated traction cable is attached to the seafaring vessel. The elastic material (for storage purposes) of the pipe supporting the whole structure is of an acoustic transparent nature [17]. Sensor spacing is kept at a constant distance with the use of centring device that supports each sensor within the pipe [17].

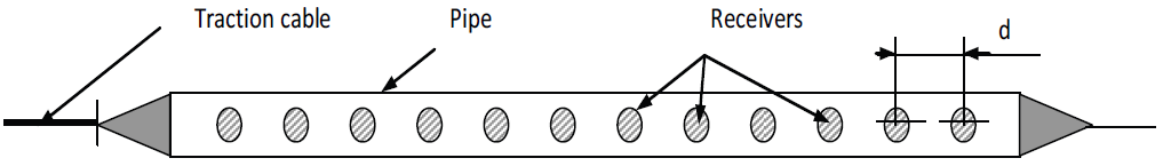

Figure 2.30: Towed linear array [17].

2.3 Dynamic Gain Control

After filtering the received signal in the required frequency band in the signal conditioner stage of the receiving end of the passive sonar, dynamic gain control is used to prepare the signal for digital conversion so that digital beamforming can be applied. Dynamic gain control for passive sonar is comprised of an Automatic Gain Controller (AGC) for each sensor channel. The purpose of it is to avoid SNR degradation when converting the signal from analogue to digital [20].

More specifically, the Signal to Quantisation Noise Ratio (SQNR) should be maximised by using an AGC. Quantisation is the process of mapping continuous amplitude to discrete amplitude. If an Analogue to Digital Converter (ADC) has *B* bits, the ADC has 2^B quantisation levels within its operating range of amplitudes. The SQNR is given by [13]:

$$SQNR = 10\log_{10}\left(\frac{P_v}{P_q}\right) dB \qquad \textbf{2.3.1}$$

- P_v - Signal power
- P_q - Quantisation noise power

It is straightforward to note that the maximum error in amplitude that can be induced by quantisation is equal to half the difference in amplitude between two quantisation levels (quantisation interval) of the ADC. This error is commonly modelled as additive white noise with zero mean distributed evenly over the interval $\left\{-\frac{\Delta}{2}\ldots\frac{\Delta}{2}\right\}$, where Δ is the quantisation interval.

Following this assumption, the quantisation noise power is given by [13]:

$$P_q = \frac{\Delta^2}{12} \qquad \textbf{2.3.2}$$

To maximise SQNR, the signal presented to the ADC should occupy as much as possible of the ADC's quantisation levels while not exceeding the operating range of amplitudes that the ADC can accommodate [20]. This will ensure that small amplitude variations are also quantised properly.

Assuming a Gaussian signal with variance σ^2 at the input of the ADC, the maximum voltage level of the signal is approximately 4σ [20]. From this, the quantisation interval size is given by $\Delta = \frac{4\sigma}{2^B}$ and the SQNR from equation 2.3.1 and 2.3.2:

$$
\begin{aligned}
SQNR &= 10\log\left(\frac{\sigma^2}{\frac{\left(\frac{4\sigma}{2^B}\right)^2}{12}}\right) \\
&= 10\log\left(\frac{3\times 2^{2b}}{4}\right) \\
&= 10[\log(3\times 2^{2b}) - \log 4] \\
&= 10[\log 3 + 2b\log 2 - \log 4] \\
&= 6{,}02b - 1{,}25\ dB
\end{aligned}
\qquad \textbf{2.3.3}
$$

Using a typical ADC of 11 bits including sign, the SQNR is equal to 58.95 dB. This means that the signal and background noise power is 58.95 dB above the quantisation noise power, which is well above the requirement of a sonar system and can be seen negligible [20].

If the assumed power of the Gaussian input signal deviates from σ^2 however, the SQNR can potentially become small even for an ADC with 11 bits. The SQNR in this case is given by

$$SQNR = 6{,}02b - 1{,}25 - 10\log\frac{\sigma^2}{\sigma_0^2}\ dB \qquad \textbf{2.3.4}$$

since the quantisation noise power is constant for any input waveform with power σ_0^2 [20]. From this, the purpose of the AGC is to ensure constant signal power at the input of the ADC to avoid fluctuating SQNR and to maximise SQNR as explained above [20].

2.4 Beamforming

In practice, the field that arrives at the array will be contaminated by noise and sources other than the one of interest. Signal processing techniques must be used to extract the desired signal from the noise and interfering sources. A time dependant signal can be temporally filtered to extract the desired signal present in a certain frequency band and suppress noise and signals occupying other frequency bands. A propagating signal is dependent on time and space and to separate the desired signal with respect to direction of propagation and frequency from interfering noise and signals, *spatiotemporal filtering* is needed. As mentioned in section 2.2.1, spatial filtering is achieved by combining omnidirectional (or directional) sensors to form an array. This is a cheaper, more flexible alternative to having a single sensor with directional characteristics as this sensor would have to be mechanically steered in a direction and if it malfunctions, all operation is lost. The process of using the directionality of an array for spatial filtering by means of array signal processing techniques is called *beamforming* [3] [6].

2.4.1 Delay-and-Sum Beamforming

Despite being the oldest and simplest array signal processing algorithm, delay-and-sum beamforming is still a viable approach today. This algorithm delays each signal incident at each sensor in the array by an appropriate amount, corresponding to a direction of reception, and adds them together [6]. The delays are the time difference of arrival of the propagating signal at each sensor. When the signal of interest is present in the direction of reception corresponding to the time delays applied to the received signals, the signal of interest will constructively interfere when adding the sensors' signals whilst the noise and signals from other directions will destructively interfere [6]. An illustration of basic delay-and-sum beamforming operation can be seen in Figure 2.31 below.

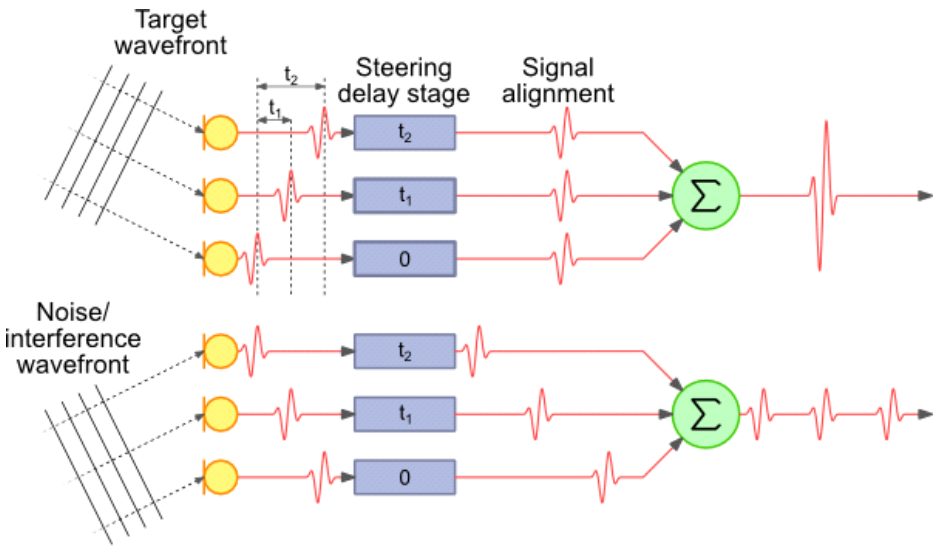

Figure 2.31: Basic delay-and-sum beamforming operation [21].

The *phase centre* of an array is the point to which all sensor positions are defined relative to and governs the time delay that is needed to focus the array's beam in a certain direction (see

section 4.3.1 for a more in depth explanation relevant to this thesis). After the steering delay stage in Figure 2.31, where each of the sensor's time delay is defined as τ_m, an amplitude weight (shading) w_m is applied to every output of each sensor and summed. If the waveform measured at the m'th sensor is $y_m(t)$, the analogue delay-and-sum beamformer's output is defined as [6]:

$$z(t) \equiv \sum_{m=0}^{M-1} w_m y_m(t - \tau_m) \qquad \textbf{2.4.1}$$

As mentioned above, when the beamformer's time delay match the direction (far-field beamforming) or position (near-field beamforming) of the signal of interest, the signal will constructively interfere and give maximum beamformer response. Thus, to find a source, all possible directions or positions must be scanned by steering the beam of the array. For far-field delay-and-sum beamforming, the time delays applied to each sensor correspond to beamforming in a certain direction (or bearing) as depicted in Figure 2.31. However, for near-field delay-and-sum beamforming, the time delays correspond to the shortest distance from each sensor to the position that is being scanned. An in depth look at how these delays are calculated will be discussed in section 4.3.1.

This thesis will focus on digital delay-and-sum beamforming. For the sampled waveform $y_m(n)$ measured at the m'th sensor, the digital delay-and-sum beamformer's output is defined as [6]

$$z(n) \equiv \sum_{m=0}^{M-1} w_m y_m(n - n_m) \qquad \textbf{2.4.2}$$

where n_m is the integer delay in samples for the m'th sensor.

All of the principles of analogue delay-and-sum beamforming discussed above are also valid for digital delay-and-sum beamforming. An important difference between analogue and digital delay-and-sum beamforming is the fact that the time delays needed to steer the beam in a certain direction is quantised to increments of the temporal sampling rate in the digital case [22]. Thus, there is a discrete set of beams that can be achieved exactly. These beams are called *synchronous beams* and the number of synchronous beam directions that can be achieved increases with an increasing temporal sampling rate [22]. It is therefore desirable with digital delay-and-sum beamforming to have a temporal sampling rate significantly higher than the Nyquist sampling rate to achieve as many synchronous beams as possible and to minimise the error in time delay introduced when asynchronous beams are formed [22]. Asynchronous beamforming can also be avoided entirely if the sampled data of each sensor is firstly zero padded and then interpolated in such a way that the exact time delay needed for a certain beam direction can be achieved but this method is outside the scope of this thesis [22].

2.5 Detection of Signals in Noise

As the name suggests, the function of the detector stage of the passive sonar system in Figure 2.1 is to detect the presence of a signal of interest in background noise. The background noise is made up of the environment noise and any other interfering signals.

The detector must make a decision between two possible hypotheses: *signal present (H_1)* or *signal absent (H_0)*. H_1 represents a signal of interest plus noise ($S + N$) and H_0 represents noise only (N).

The decision itself is denoted by D_0 and D_1 for H_0 and H_1 respectively. These decisions can be correct or incorrect in both cases and thus creates four different scenarios. A summary of these scenarios is presented in the *decision matrix* below:

	Decision	
True Hypothesis	**Signal Present (D_0)**	**Signal Absent (D_1)**
Signal Present - H_1 is true	*Correct detection*	*Miss*
Signal Absent - H_0 is true	*False Alarm*	*Null Decision*

Table 2.1: Binary decision matrix for a passive sonar detector [3]

The probability that a *correct detection* is made is referred to as the *detection probability* and is given by [3]:

$$p(D) = p(D_1|H_1) \qquad \textbf{2.5.1}$$

From this equation the probability of a *miss* is $1 - p(D)$.

The probability that a *false alarm* will occur is given by [3]:

$$p(FA) = p(D_1|H_0) \qquad \textbf{2.5.2}$$

From this equation the probability of a *null decision* is $1 - p(FA)$.

2.5.1 Detection Threshold and Detection Index

To make the decision whether the signal at the input of the detector is signal plus noise or noise only, a threshold is needed. Above this threshold H_1 is chosen and below this threshold H_0 is chosen. This threshold is called the *Detection Threshold (DT)* and is defined as the ratio of the signal power in decibels to the noise power in decibels as measured in the sensor bandwidth at the input of the sensor, required for detection at some preassigned level of correctness of the detection decision [23]. Here the input to the "receiver" is the same as the input to the detector stage in the passive sonar system flow diagram in Figure 2.1 [23].

In equation form, it is given by [1]:

$$DT = 10\log\left(\frac{S}{N}\right)_{in}\ dB \qquad \textbf{2.5.3}$$

- S - Signal power in the sensor bandwidth
- N - Noise power in the sensor bandwidth

The detection threshold should be chosen so that an acceptable balance is reached between false alarm ($p(FA)$) and detection ($p(D)$) probabilities. If the threshold is set too high, a *miss* will occur too often ($p(D)$ becomes too low) whereas if the threshold is set too low, many false alarms will occur ($p(FA)$ becomes too high). Assuming Gaussian probability density (this assumption is valid for a large number of samples according to the Central Limit Theorem [23]) functions for both signal plus noise and noise only (where the distributions for a single sample y of the random process is given by $p_{S+N}(y) = p(y|H_1)$ and $p_N(y) = p(y|H_0)$ [3]) with equal variance σ^2 [1] and plotting them in Figure 2.32 below, the above explained occurrence can be graphically interpreted with $p(D) = P_d$ and $p(FA) = P_{fa}$ in the figure. Notice that when the threshold level is increased, both P_d and P_{fa} decreases, making the chance for a "miss" more likely. When the threshold level is decreased, both P_d and P_{fa} increases, making the chance for a false alarm more likely.

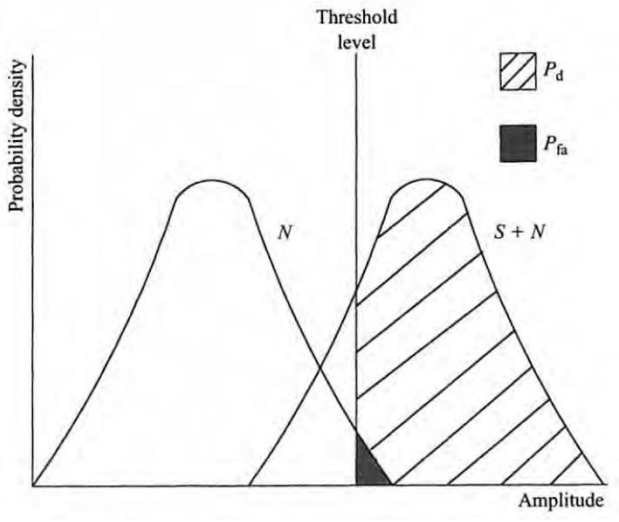

Figure 2.32: Probability density functions of signal plus noise and noise only with threshold level indicated [24].

To choose a satisfactory detection threshold, a *Receiver-Operating-Characteristic* (ROC) curve is used to obtain the detection index for a specified $p(D)$ and $p(FA)$ which in turn gives us the detection threshold. A ROC curve is a plot of $p(D)$ against $p(FA)$ for a certain *detection index d.* The detection index is a measure of the distance between two probability distribution functions [3] and thus an indication of how easy a signal of interest can be detected in noise.

The detection index is defined as [1]:

$$d = \frac{[M_{S+N} - M_N]^2}{\sigma^2}, \qquad \textbf{2.5.4}$$

where M_{S+N} and M_N are the mean of the signal plus noise and noise only Gaussian distribution functions respectively. Figure 2.33 shows ROC curves for different detection indices on probability coordinates assuming Gaussian distributions for signal plus noise and noise only as explained above.

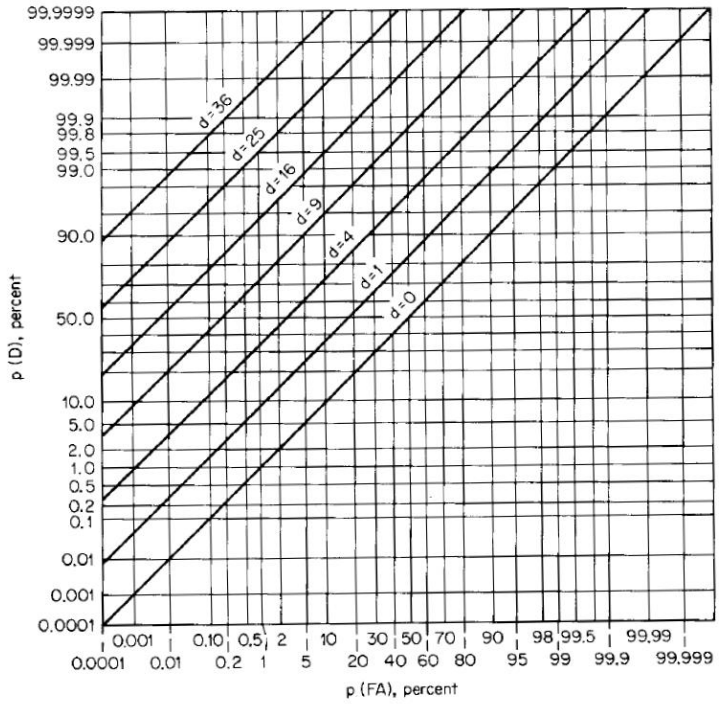

Figure 2.33: Receiver Operating Characteristic curve for different detection indices [3, p. 236].

The most efficient and most commonly used detector for a broadband passive sonar, assuming an unknown signal in noise, is the square law (power) detector [3] [23]. Broadband detection compares the output of the passive sonar beams to make a detection decision whereas narrowband detection examines the content within each beam to make a detection decision [23]. The equation for the detection index *d* of a broadband power detector is given by [3]:

$$d = B \cdot T \left(\frac{S}{N}\right)_{in}^2 \qquad \textbf{2.5.5}$$

- B - Bandwidth of input filters
- T - Integration time
- S - Signal power at the sensor input, measured in the sensor bandwidth
- N - Noise power at the sensor input, measured in the sensor bandwidth

From equation 2.5.3 and the expression for the detection index above, the detection threshold is obtained [23]:

$$DT = 10\log_{10}\left(\frac{S}{N}\right)_{in} = 10\log_{10}\left(\frac{d}{BT}\right)^{\frac{1}{2}} = 5\log_{10}\left(\frac{d}{BT}\right) \qquad \textbf{2.5.6}$$

- d - Detection index
- B - Bandwidth of input filters
- T - Integration time

It is worth noting that when a broadband power detector is used, which presents significant signal and noise fluctuation within the receiving band, B should be calculated as the equivalent noise bandwidth given by [23]:

$$B_{eff} = \frac{(\int noise_{1\,Hz}(f)\,df)^2}{\int (noise_{1\,Hz}(f))^2\,df}, \qquad \textbf{2.5.7}$$

where $noise_{1\,Hz}(f)$ is the noise power in a 1 Hz band at frequency f and the integrals are calculated over the full bandwidth of the sonar [23].

Also, note that the ROC curve in Figure 2.33 is only valid for Gaussian signal plus noise and noise only distributions. For large sample sizes (bandwidth-time product, $BT > 100$) the Gaussian distribution assumption is valid according to the Central-Limit Theorem as mentioned previously [23]. For smaller bandwidth-time products ($BT \ll 100$), a correction factor must be added to the detection threshold (see Figure 2.34) or the correct distribution must be used to obtain the corresponding ROC curve (an exponential distribution in the case of a power detector) [23].

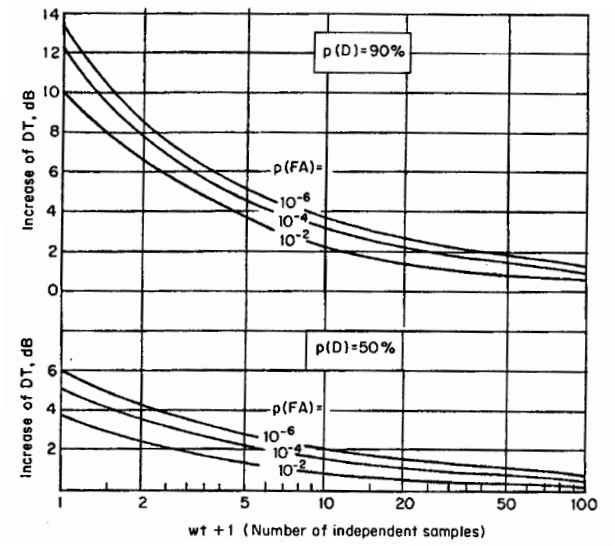

Figure 2.34: Plot of the increase in detection threshold as a function of the bandwidth-time product for various false alarm and detection probabilities [23]

2.6 Operator's Panel Displays and Controls

After computerised signal processing, relieving the operator from repetitive and tiring tasks, the results are usually presented to an operator to make final decisions and interpretations with respect to the signals received [3]. An overview of commonly used means of displaying passive sonar data is presented here.

The **time-bearing display** shown in Figure 2.35 is used for broadband power detection [24]. Signals labelled as targets (signal of interest) by the detector at a certain bearing (or beam of the beamformer) are shown as dots on the display for a certain time frame. These dots form lines as time progresses and certain derivations of movement and type of target can be made from this information.

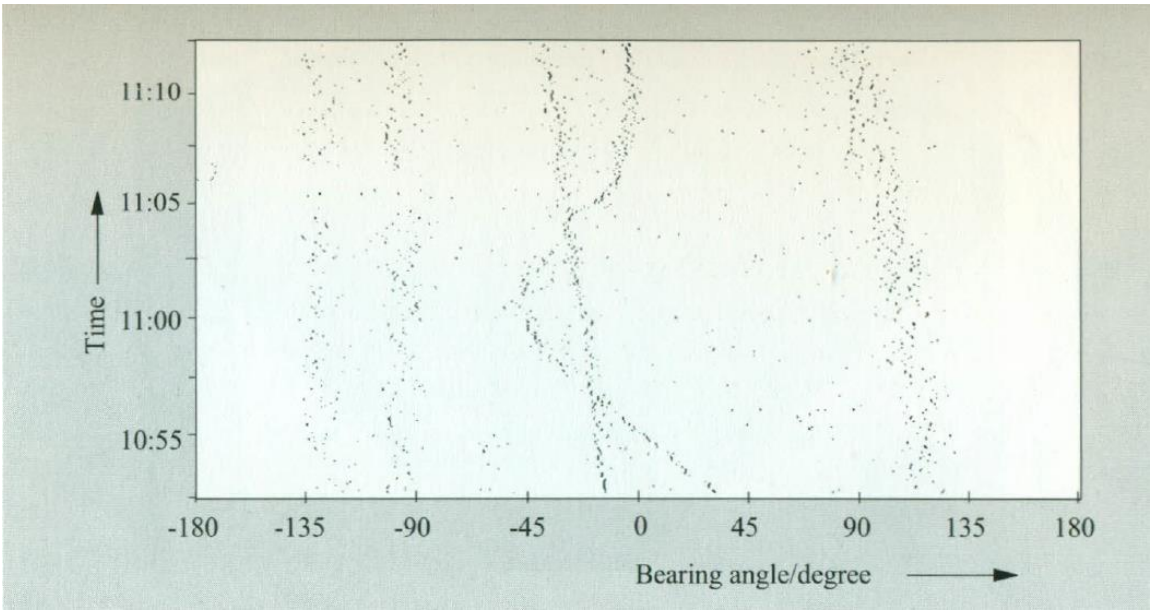

Figure 2.35: Time-bearing display showing targets as dots which forms the lines seen in the image as time progresses [3]

The LOFAR display shown in Figure 2.36 is used for narrowband power detection. Frequency, using the short time Fourier transform for a certain time frame, is plotted against time and is in effect a spectrogram. Only one target (or beam) in a certain direction can be displayed at a time. The operator can classify targets and perform motion analysis of the target using this type of displayed information [24].

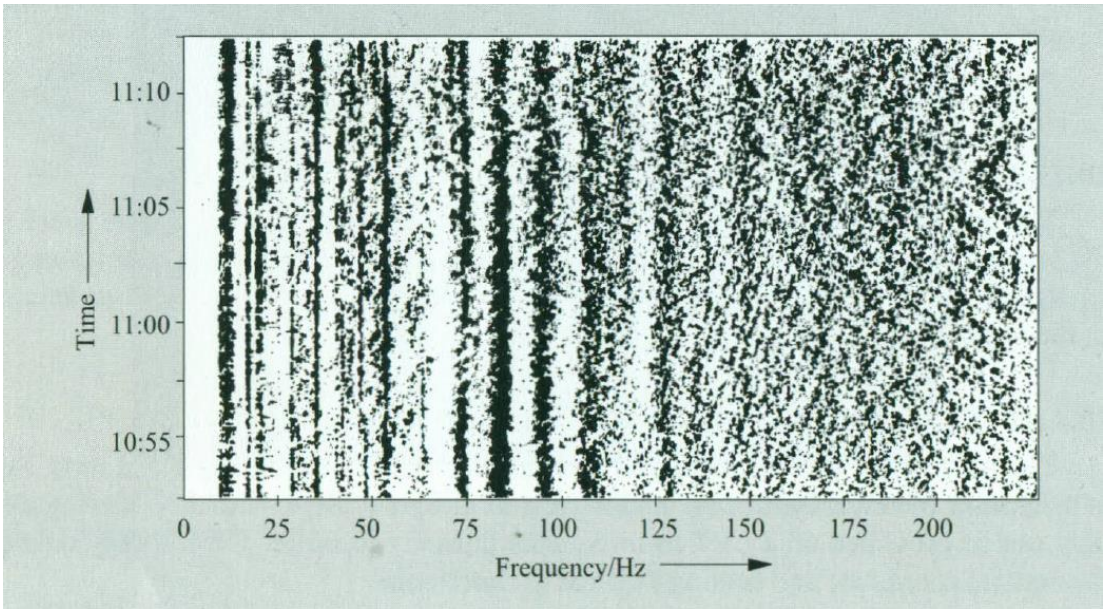

Figure 2.36: LOFAR display which shows frequency vs time of one target [3]

The frequency-bearing display shown in Figure 2.37 is used for narrowband detection or broadband detection. The local maxima, corresponding to a certain frequency, of the short time power spectra of the signal at each bearing (or beam) are displayed as dots with varying size depending on the intensity. Dots on the frequency-bearing display at the same bearing come with high probability from the same target [3].

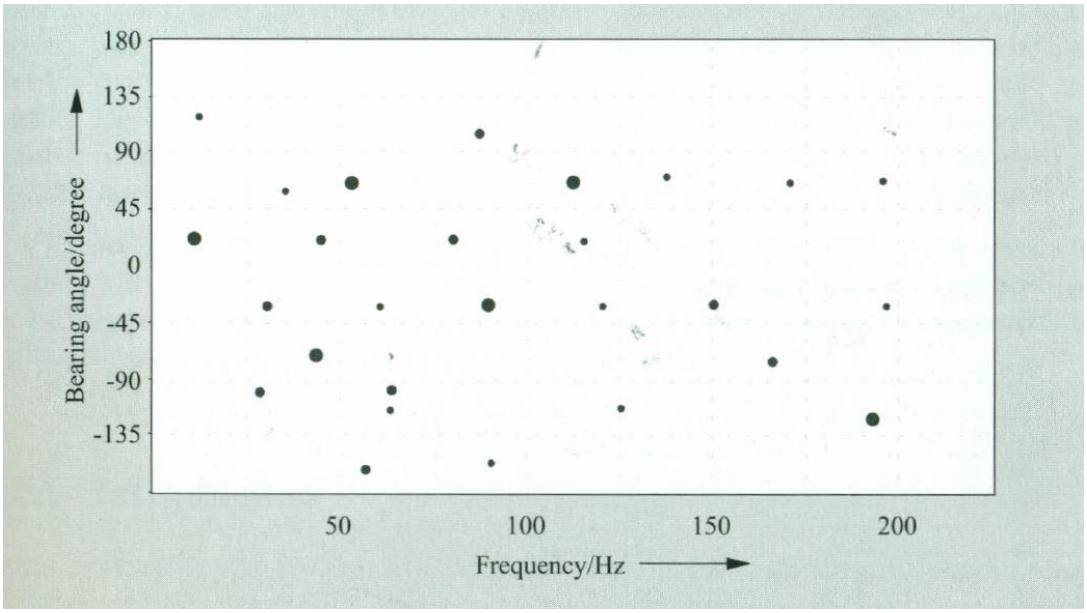

Figure 2.37: Frequency-bearing display with the dots representing the local maxima of the short time power spectra at different frequencies [3]

2.7 The Passive Sonar Equation

The passive sonar equation is used for performance prediction and design of a passive sonar system. It is comprised of all the *sonar parameters* that have an effect on the SNR at the input of the sensor. The equation can be solved for a certain unknown, which can be a parameter of performance prediction (e.g. transmission loss from which operating range can be calculated) or a parameter needed for the design of the system (e.g. the directivity required for a desired range of detection) [1]. Certain combinations of the sonar parameters can also be used for performance prediction [1]. The sonar system parameters are determined by the system's equipment, the medium in which it operates and the nature of the target of interest [1]. Each of these parameters is levels in decibel units relative to the standard reference intensity of a 1µPa RMS pressure plane wave. Assuming the passive sonar is used for detection, when the input SNR is above a certain detection threshold the decision is made that a target is present and vice versa. When the detection threshold equals the input SNR, the passive sonar equation is given by [1]:

$$DT = SL - TL - (NL - DI) \qquad \textbf{2.7.1}$$

Equipment parameters:

- *DT* - *Detection Threshold* in the sensor bandwidth.
- *DI* - *Directivity Index* of the array at range *r*.

Medium parameters:

- *NL* - *Noise Level* in the sensor bandwidth at the sensor location.
- *TL* - *Transmission Loss* of the acoustic intensity over the propagation path referenced at 1 m from source and sensor.

Target parameters:

- *SL* - Target *Source Level* of the radiating source in the sensor bandwidth referenced at 1 m distance from the source.

Combinations of the Sonar Parameters for Passive Sonar		
Performance Figure	$SL - (NL - DI)$	Difference between the source level and the noise level measured at the sensor terminals
Figure of Merit	$SL - (NL - DI + DT)$	Maximum allowable one-way transmission loss

Table 2.2: Combinations of the sonar parameters for passive sonar [1].

Chapter 3
Compute Unified Device Architecture

This chapter will present an overview of the principle operational structure of NVIDIA's Compute Unified Device Architecture (CUDA). Wherever possible, the overview will be restricted to CUDA functionality that was used in this study. A more in depth discussion can be found in section 4.4 where CUDA programs specific to this study will be explained.

3.1 Background

This section was adapted from "CUDA by Example: An Introduction to General-Purpose GPU Programming" by Sanders, Jason, Kandrot, and Edward [25] unless otherwise stated.

One of the most prominent improvements made to Central Processing Units (CPU's) since the first personal computers were released in the early 1980's was the clock speed of the CPU. Increasing from a mere 1 MHz to between 1 GHz and 4 GHz over the past 30 years or so, the clock speed of the CPU was the driving force for CPU performance improvement. Unfortunately, this soaring clock speed was met with considerable resistance from power and heat restrictions as well as the approaching physical limit of transistor size and drove the industry to search for improvement elsewhere.

Although parallel processing is fairly new to personal computers, supercomputers have been using multiple processors for decades. This was the logical next step in personal computer CPU performance improvement and gave rise to the multicore revolution.

Parallel to this revolution came an increasing interest in performing general-purpose computations on Graphics Processing Units (GPUs). With the release of programmable pipelines on the GPU through the DirectX 8.0 standard, researchers began using graphics Application Programming Interfaces (APIs) such as OpenGL and DirectX to perform General Purpose Computations on GPU's (GPGPU). Although extremely complex, researchers achieved this by making their applications seem like traditional rendering applications to the GPU. Initial results of these applications were promising due to the high arithmetic throughput of GPU's. Constraints such as unpredictable floating-point data handling, limited memory management options, poor debugging methods, and the sheer complexity of writing general purpose computations code in graphics-only programming languages called for a new API and a change in GPU architecture.

NVIDIA sought to solve these problems with the introduction of their new CUDA architecture and accompanying language and API, a modified version of industry standard C, called CUDA C. This new architecture allowed the developer to control each Arithmetic Logic Unit (ALU) on the GPU. These new ALUs also adhered to the IEEE standards for single-precision floating-point arithmetic and were designed to use an instruction set that better suits the needs of GPGPU. Memory management options were also expanded. With all these improvements CUDA C became the first high-level programming language specifically tailored for GPGPU

and was released in 2007. CUDA also supports other languages such Fortran and OpenCL. A complete list of supported languages can be found in the NVIDIA CUDA C Programming Guide [26]. This thesis will focus on CUDA C.

3.1.1 The GPU as a General-Purpose Computations Processor

Since its inception the GPU has evolved into a processor that is highly parallel, multithreaded, and has many cores which produces immense computational power and has a very high memory bandwidth [27] which can be seen in Figure 3.1 and Figure 3.2. This evolution was mainly driven by the demand for real-time, high definition 3D graphics [27].

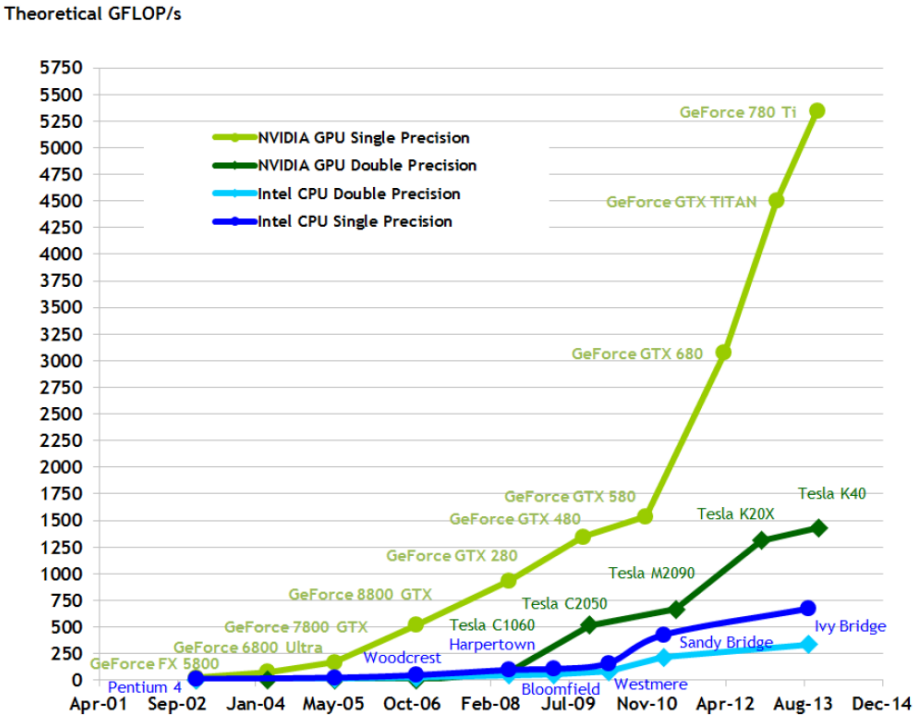

Figure 3.1: FLoating-point Operations Per Second (FLOPS) for the CPU and GPU [26].

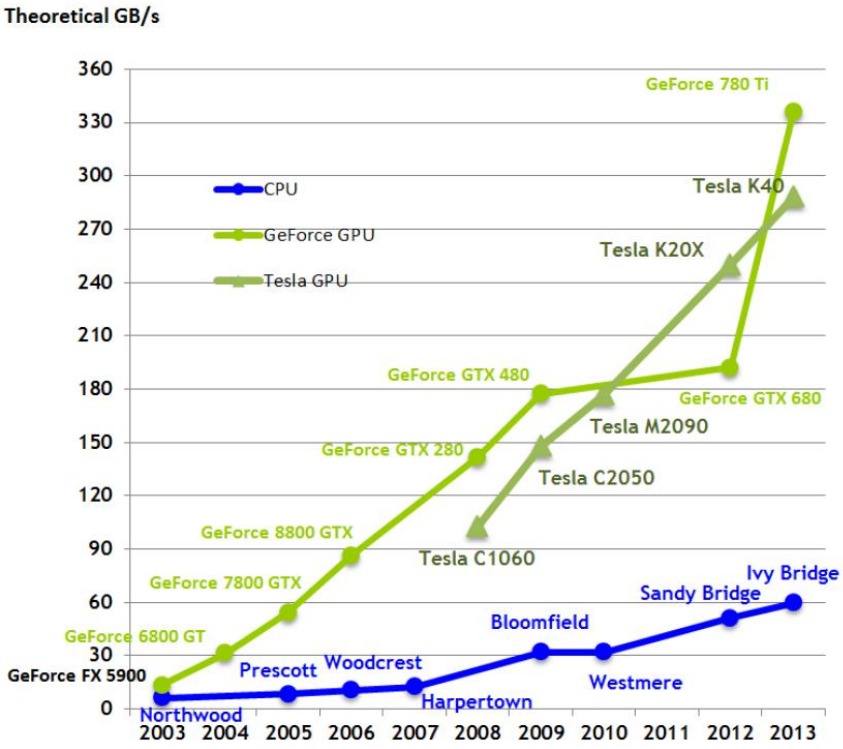

Figure 3.2: Memory bandwidth for the CPU and GPU [26].

The trend illustrated above can be explained by the fact that the GPU is specialised for compute intensive, highly parallel computation [27]. It achieves this using many more transistors devoted to data processing than the CPU, where data caching and flow control is usually of higher importance [27] and is illustrated in Figure 3.3.

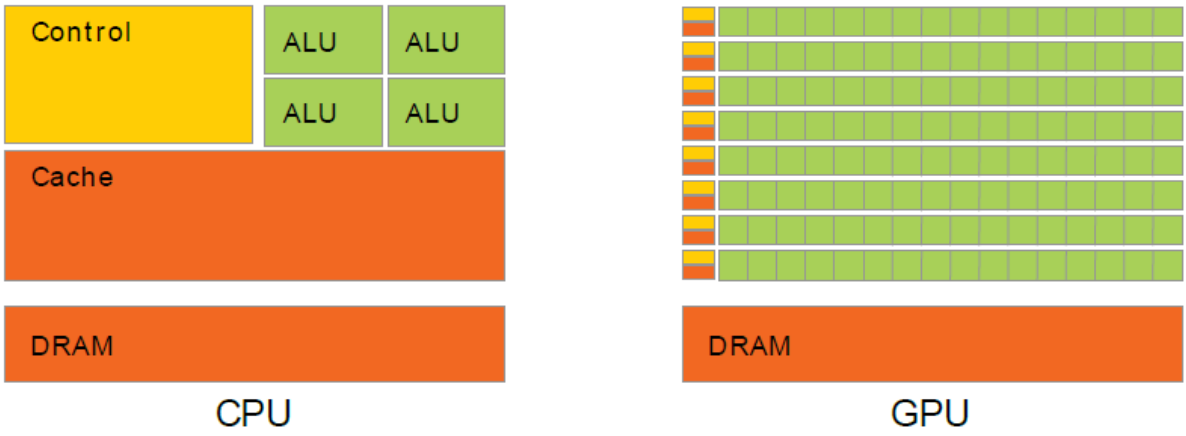

Figure 3.3: GPU and CPU comparison of the number of ALU's [26].

The highly parallel computation mentioned above is, more specifically, data-parallel computations where one set of data is used for an algorithm that executes the same task in parallel but on different sections of the data set.

3.1.2 CPU and GPU Interaction within a CUDA C Program

The CUDA programming model views the CPU as the *host* and the GPU (or multiple GPU's) as the *device* of the CUDA C program where the device is seen as an external coprocessor for the host [27]. A kernel is a function (similar to a C function) that executes on the GPU. As with any other C program the "main" or any other standard C functions of a CUDA C program executes serially on the CPU and can make a call to a kernel to run parallel computations on the GPU. When the kernel is launched, control is immediately returned to the host irrespective of the completion of the kernel's tasks [27]. It is also possible to execute two or more kernels concurrently as well as performing memory copies between the host and the device concurrently with these kernels (refer to section 3.4) [27].

3.1.3 CUDA - A Scalable Programming Model

CUDA has a scalable programming model which means that applications written in CUDA will automatically scale the parallelism of the application according to the number of GPU Streaming Multiprocessors (SM's) and the number of cores available within these SM's (depending on which GPU is used) [27].

There exist three core abstractions that govern the scalability of the application, while maintaining a low level of complexity for the programmer by adding minimal language extensions (e.g. extensions to C) [27]:

- **A hierarchy of thread groups (see section 3.2)**
- **Shared memories (see section 3.3)**
- **Barrier synchronisation (see section 3.3 for thread synchronisation and section 3.4 for task synchronisation)**

These abstractions, along with other important CUDA features will be explained in the sections that follow.

3.2 Thread Hierarchy

A thread resides in the lowest level of the hierarchy. One thread can be seen as one iteration of a computation on one piece of data within a data set. For example, if two arrays of data are added together according to $C[x] = A[x] + B[x]$, then the computation $C[0] = A[0] + B[0]$ would be carried out by one thread.

Thread groups are referred to as a *block (a block of threads)*. Each block can be one-dimensional, two-dimensional, or three-dimensional, and each thread within a block is identified by its one-dimensional, two-dimensional, or three-dimensional *thread index* [27]. This representation of blocks presents a natural way of dealing with one-dimensional (e.g. vector data), two-dimensional (e.g. matrix data), or three-dimensional (e.g. volumetric data) data.

Block groups are referred to as a *grid (a grid of blocks)* and resides in the highest level of the hierarchy. In the same way that blocks do, grids can occupy three dimensions at most and each block in the grid is identified by its block index as explained above for blocks of threads [27]. See Figure 3.4 for an illustration of the thread hierarchy.

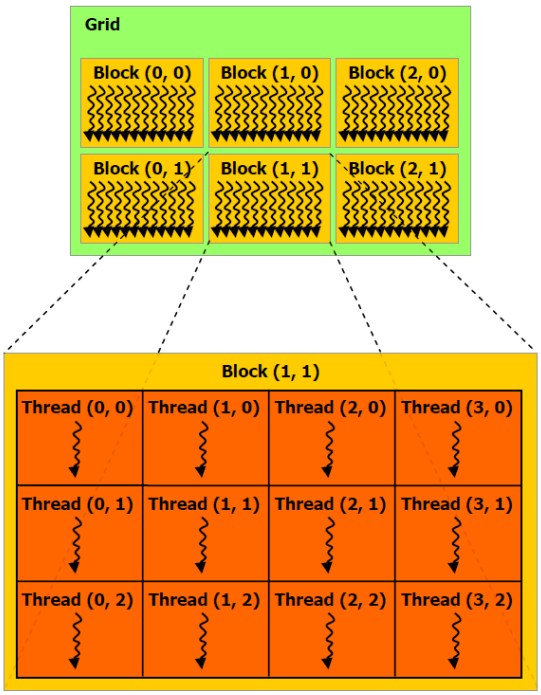

Figure 3.4: The hierarchy of thread and block groups in CUDA [27].

The blocks within a grid and the threads within a block are specified by the user when calling a kernel. With this hierarchy of threads, blocks, and grids, the runtime system distributes the blocks according to the number of SM's on the GPU. This runtime distribution is done in a manner that makes it possible to run the same code on a different GPU that will result in an

increased (less SM's) or decreased (more SM's) computation time. See Figure 3.5 for an illustration of this distribution process.

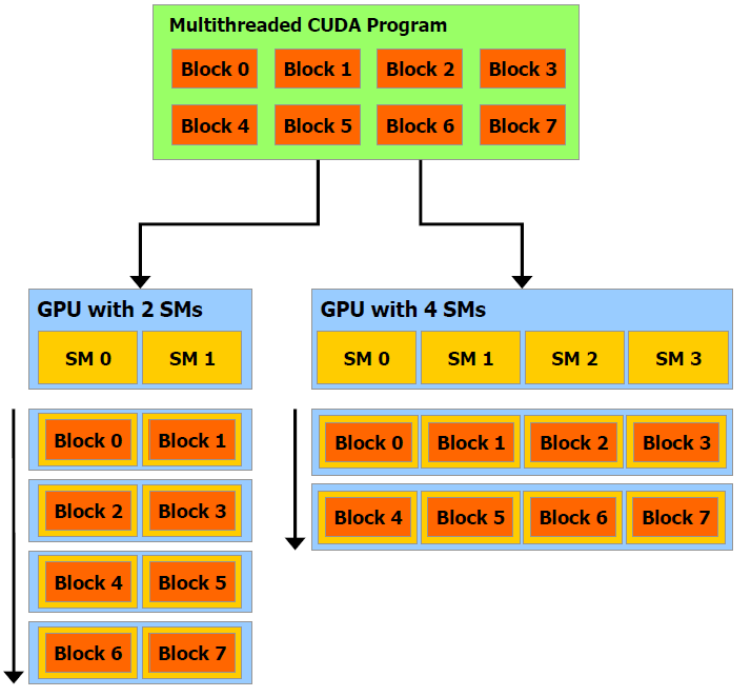

Figure 3.5: Illustration of the scalability of CUDA programs

When blocks are distributed to the available SM's, each SM partitions them further into warps. A *warp* is a group of 32 parallel threads. The SM creates, manages, schedules, and executes these warps on the available cores of the SM [27].

Each warp contains threads with consecutively increasing thread indices. Full efficiency with respect to parallelism is achieved when all threads within a warp agree on their execution path so that one common instruction at a time can be executed across all the threads within a warp. Data-dependant conditional statements can cause threads within a warp to diverge to different execution paths. These paths will be executed serially until all paths have been executed. This is called warp divergence and should be avoided where possible. For the same reason, it is also recommended to choose the number of threads per block as a multiple of warp size. [27]

3.3 Memory Hierarchy

The CUDA programming model separates *host memory* and *device memory* [27]. A CUDA program manages both these memories by allocating or deallocating device memory (visible to kernels) as well as managing memory copies between host and device.

The CUDA programming model breaks down different types of memories into the following three main groups (see Figure 3.6):

- **Global Memory:** Global memory resides in the Dynamic Random Access Memory (DRAM) of the device. As the name suggests, all threads executing on the GPU can access global memory and it is persistent across kernel launches within the same application [27].

  Compared to shared memory (explained below), global memory has a higher latency and lower bandwidth. Global memory use should always be replaced by shared memory when possible [27].

- **Per-block shared memory:** Variables can be defined in a kernel to reside in the shared memory of the GPU to store data that is accessed by multiple threads within a block for some computation or to store results of a computation. Shared memory is an on-chip memory that is equivalent to a user-defined cache [27]. As mentioned above, it has a much lower latency and much higher bandwidth than local (explained below) or global memory [27].

  A block of threads needs to have its own block of shared memory declared as shared memory for one block of threads is inaccessible to the other blocks of threads [27]. Threads within a block can be synchronised when they need to share results or overwrite data within shared memory with the results of a computation. The synchronisation point is user defined and forces the kernel to wait until all threads within the block is finished executing their tasks [27].

- **Per-thread local memory:** Each thread's local memory is comprised of its own registers and local memory. Registers are used to store small amounts of data for fast access by the processor when performing instructions. Local is subject to the same high latency and low bandwidth that memory accesses have in global memory [27].

  When declaring automatic variables within a kernel such as dynamic arrays or large arrays or structures (consuming too much register space), the compiler will likely place these variables in local memory [27]. Furthermore, when a kernel uses more registers than available, the data will reside in local memory and is called *register spilling* [27].

Local memory storage should be avoided for maximum performance [27].

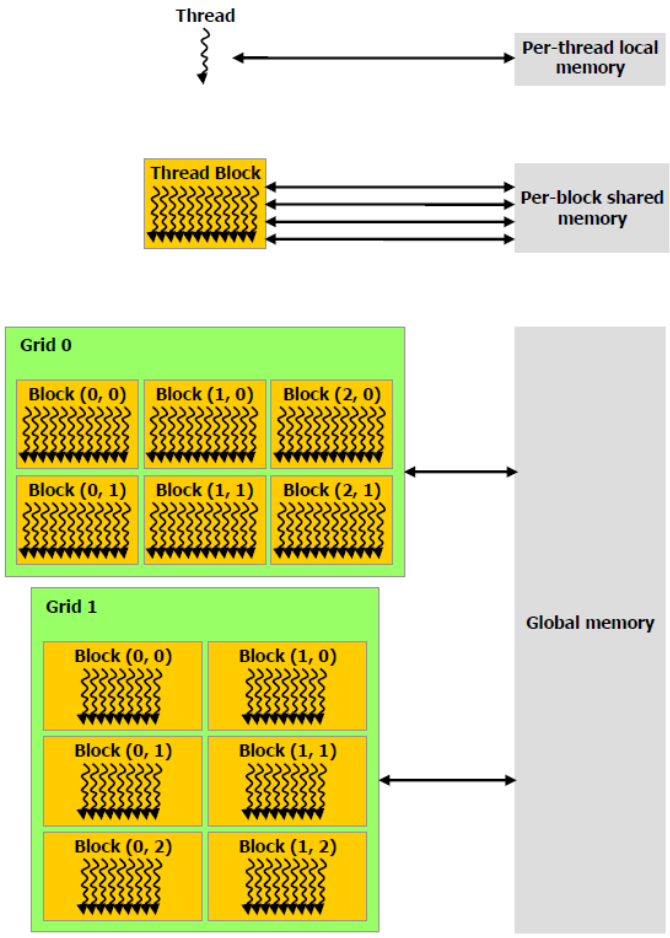

Figure 3.6: Memory hierarchy of the CUDA programming model [27].

3.4 Streams

As shown in section 3.1.1, the GPU is superior to the CPU when computations are of a data-parallel nature. Another type of parallelism, *task-parallelism*, can be achieved with CUDA. Task-parallelism is the parallel execution of two or more completely different tasks [25]. Currently this type of parallelism is better supported and more flexible on a CPU than a GPU but possible nonetheless with the potential of speeding up applications even further [25].

Streams and certain function calls that are *asynchronous* are used in CUDA to facilitate concurrent execution of tasks [27].

A stream can be thought of as a set of tasks that were issued to the stream by the user in a certain order. The device then executes these tasks in the order that they were issued to the stream [25]. Multiple streams can be launched concurrently with the potential of parallel execution of tasks between different streams [25].

Asynchronous function calls, as opposed to *synchronous* function calls, places a request to perform a task on the device and immediately returns to the host without the guarantee that the task is completed whereas synchronous function calls returns only when the task is complete [27].

CUDA supports concurrency for the following tasks [27]:

- **Concurrent execution between host and device** – As mentioned above, certain function calls are asynchronous and returns control to the host once the request to perform a task on the device is placed. Concurrent execution between the host and device can be achieved by using the following asynchronous function calls [27]:
  - Kernel launches
  - Certain memory copies between the host and the device or between different address spaces of the same device memory
  - Memory set function calls

  When control is returned to the host after the request to perform a task is placed the host is free to complete another task while the device completes the requested tasks [27].
- **Data transfer and kernel execution concurrency** – Concurrent kernel execution and data transfers between page-locked host memory and device memory. Page-locked host memory's residency will always be in physical memory and the operating system will never page it out to disk [25]. This feature enables the GPU to use Direct Memory Access (DMA) to copy data to the host and vice versa without intervention from the CPU and enjoys superior performance [25].
- **Concurrent kernel execution** – Multiple kernels can be executed concurrently given that they are from the same context.
- **Concurrent data transfers** – Data transfers from page-locked host memory to device memory concurrently with data transfers from device memory to page-locked host memory.

Synchronisation of asynchronous streams is important to ensure that all tasks are completed within the stream before memory reads or memory copies of the results are made. Streams can be synchronised implicitly or explicitly.

Explicit synchronisation is achieved with the use of functions that CUDA provides. With these functions, it is possible at a user-defined point to synchronise all the streams running on the device, synchronise a specific stream, synchronise a stream when an event has completed, or query a stream about its current status [27].

Implicit synchronisation occurs when certain operations such as certain memory allocations, a device memory set, certain memory copies, or dependency checks are issued between two streams by the host [27].

Synchronisation methods should be used carefully and delayed as long as possible to avoid unnecessary slowdowns in applications [27]. A full explanation of these synchronisation methods can be found in the NVIDIA CUDA C Programming Guide [27].

Chapter 4 Simulations and Results

This chapter presents the simulations and results generated to meet the outcomes of this project. The complexity of passive sonar systems combined with the lack of any system specifications for this project shifted the focus from hardware implementation to simulations. A discussion of basic sonar simulation principles will pave the way for signal simulations (including underwater ambient- and ship noise), delay-and-sum beamforming simulations in MATLAB, and delay-and-sum beamforming simulations done in CUDA.

4.1 Overview of Sonar Simulation Principles

The simulation of a sonar system can be divided into three main categories: environment simulation, signal processing simulation, and post processing simulation. Refer to Figure 4.1 for a more in depth breakdown of each. Note that the subsections of these categories are for passive and active sonar.

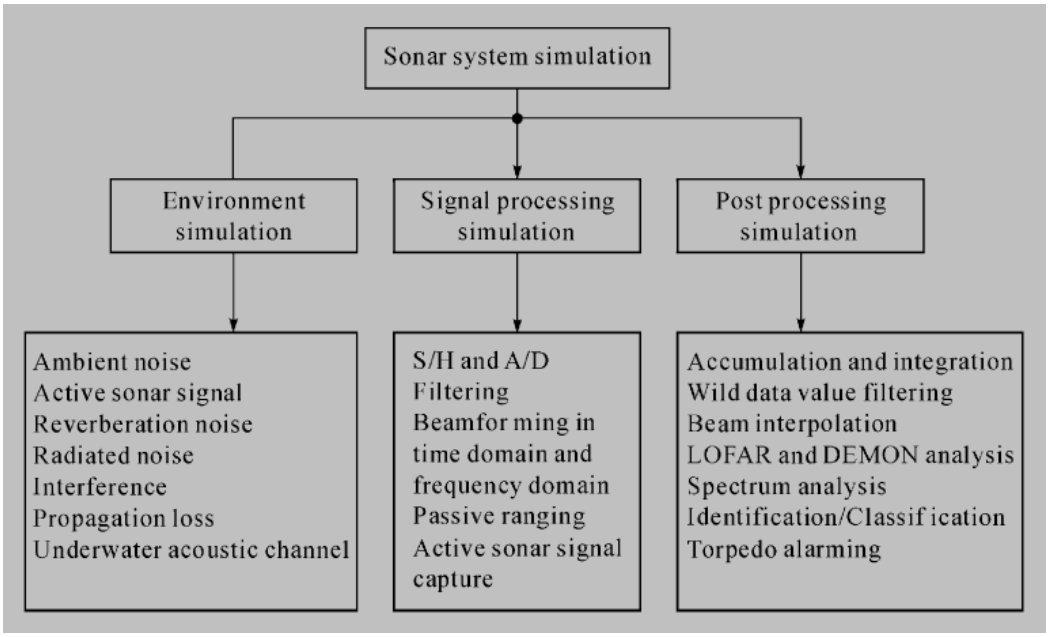

Figure 4.1: Sonar system simulation breakdown [5]

Environment simulation simulates the types of signals present and behaviour thereof in the environment where the sonar operates. The other two categories, signal processing simulation and post processing simulation, simulates different aspects of the sonar itself. The subsequent sections will focus on each of these main categories, highlighting a selected few sub categories.

4.2 Signal Simulations

Signal simulations falls in the category of environment simulation in Figure 4.1. This section will only focus on the radiated noise (section 4.2.1) and ambient noise (section 4.2.2) subsections of environment simulation and combine these two to form the signal that is used in the beamforming simulations.

The simulation of an underwater environment is complex with many interacting variables, whilst also having an influence on each other. It is outside the scope of this thesis to study all of the different types of noise that are found in the sea and the non-homogeneous medium through which they propagate. Rather, ideal conditions are assumed and discussed in the following sections.

4.2.1 Radiated Noise of a Source

The source in this context is a target of interest. Passive sonar is used for detection and tracking of a number of different types of sources including the well-known cases of detecting surface sea fearing vessels, submarines, and torpedoes as well as the lesser known case of detecting divers. The noise from a surface ship, submarine, and torpedo mainly has three contributors [5]:

- **Mechanical noise**: This noise consists of mechanical vibrations emanating from the main engine, auxiliary machines and various parts of air equipment.
- **Propeller noise**: The beating of the propeller in water and the vibration of the propeller shaft causes this noise.
- **Water dynamic noise**: The friction of the vessel's hull with water and the resonance of the hull caused by movement constitute this type of noise.

For this thesis, the radiated noise of a submarine at periscope depth and at a speed of 4 knots in the band of 1 to 10 kHz was simulated. In this frequency band, its spectrum has an approximate linear decay of -20 dB per decade as can be seen in Figure 4.2. This frequency band is chosen for two reasons:

- Most sea faring vessels has an approximate linear decay in this band on a log scale frequency spectrum that makes it convenient to simulate [5] [3] [1] [8]. It is common practice to assume this linear decay when simulating a source with these characteristics.
- More importantly, to avoid spatial aliasing, the spacing between sensors in the passive sonar array should be $d = \frac{\lambda}{2}$ or less (as discussed in section 2.2.5). Due to this fact, for a passive sonar array to fit on an AUV with the most sensors possible (which translates to a higher SNR), the higher the upper frequency in the band of operation, the better.

  As an example, consider a passive sonar array operating in the band of 100 to a 1000 Hz. The wavelength of a 1 kHz signal traveling through the sea, assuming a speed of 1500 m/s, is equal to $\lambda = \frac{v}{f} = \frac{1500}{1000} = 1.5\ m$. The spacing needed between sensors is then $\frac{1.5}{2} = 0.75\ m = 750cm$ which for standard size AUVs is unrealisable in the sense

that only a small amount of sensors can be mounted on the relatively small body of an AUV.

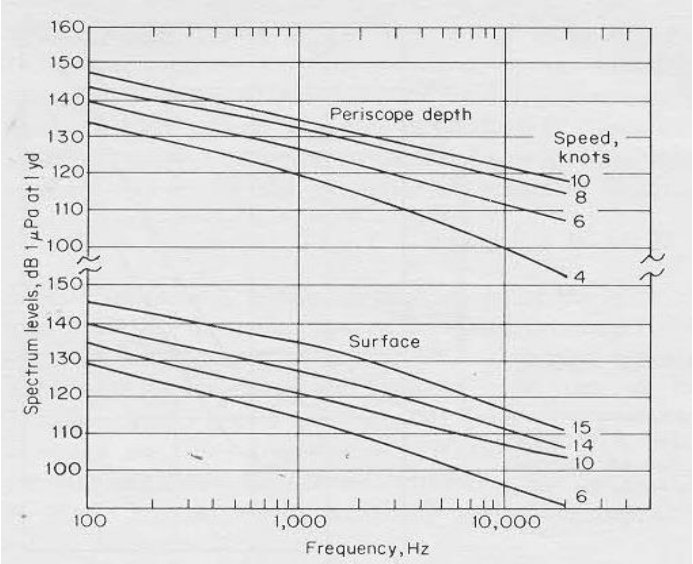

Figure 4.2: Smoothed spectra of a submarine on electric drive at periscope depth and on the surface with different speeds [1]

As mentioned in section 2.5.1, signal plus noise (source plus noise) and noise only can be assumed to have a Gaussian probability density function. The source can thus be approximated by filtering Gaussian white noise with a filter having a -20 dB per decade decay magnitude response after which a bandpass filter is used to filter in the band 1 to 10 kHz with 0 dB magnitude response in the band.

To design the first filter, MATLAB and its signal processing toolbox are used to write an algorithm that receives the desired form of a filter (defined by a number of points on its magnitude frequency response) and then recursively design an Infinite Impulse Response (IIR) filter using the least-squares method. This was done using MATLAB's "yulewalk" function.

To compensate for the phase distortion introduced by filtering with an IIR filter, the designed IIR filter is used to filter the data in the forwards direction and then used to filter the reverse of this filtered sequence again. The time reverse of the second filtering operation output is then used. This output then provides data with the desired magnitude response squared, and with zero phase distortion. By filtering twice, the magnitude of the data is modified by the square of the filter's magnitude response. These operations are carried out by MATLAB's "filtfilt" function.

Thus, the desired filter is one with a 0 dB response in the band 0 to 0.1 kHz and a gradient of -10 dB per decade (to achieve -20 dB per decade after filtering the data) in the band 0.1 to 100 kHz. This filter's magnitude response can be seen in Figure 4.3.

.

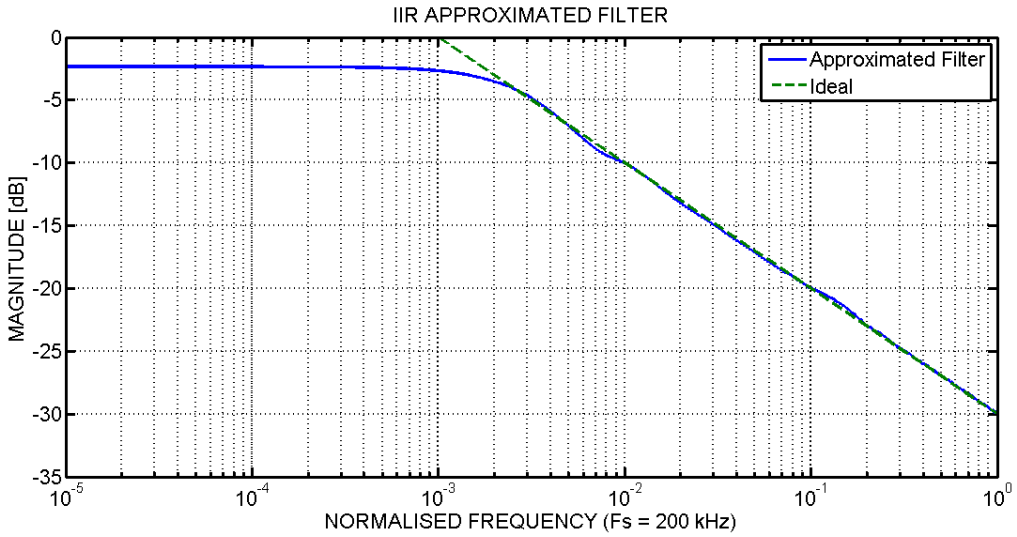

Figure 4.3: Frequency response of the approximated IIR filter for source simulation.

Figure 4.4 shows the single-sided power spectral density (PSD) spectrum of the filtered Gaussian white noise with -20 dB per decade decay.

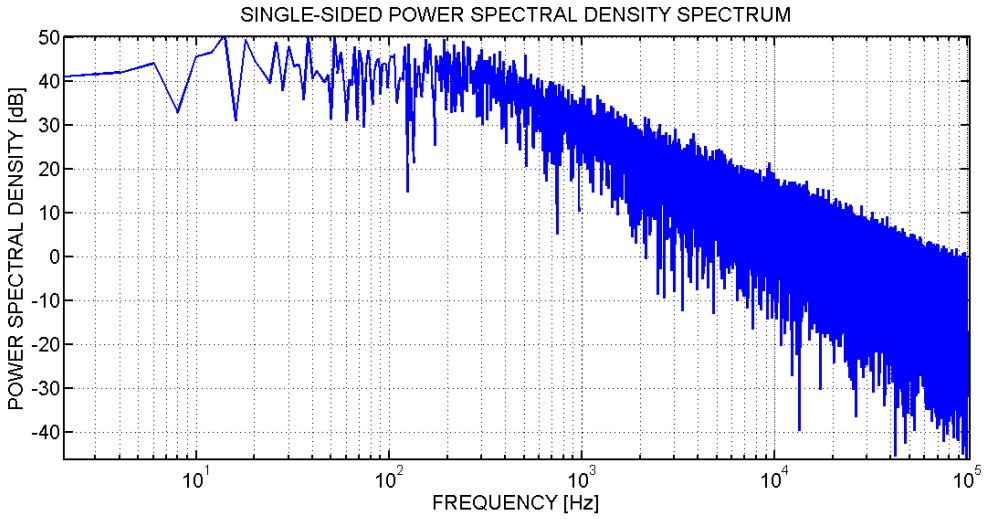

Figure 4.4: Single-sided power spectral density spectrum of filtered Gaussian white noise for source simulation.

To see the -20 dB per decade decay of the filtered Gaussian white noise clearly, the average of every N samples in the PSD spectrum of Figure 4.4 was calculated. With N = 100, the average PSD spectrum is shown in Figure 4.5.

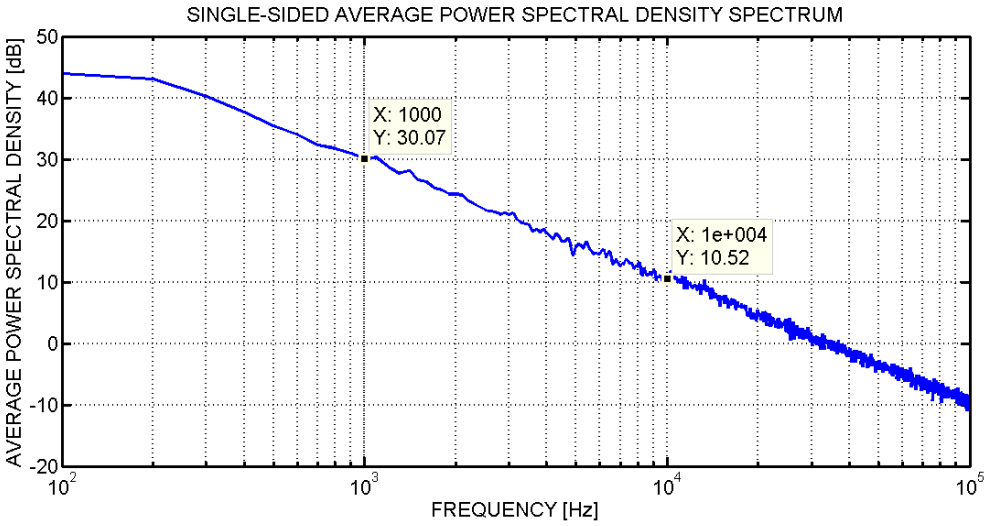

Figure 4.5: Single-sided average power spectral density spectrum of Figure 4.4

To filter the data of Figure 4.4 in the desired frequency band of 1 to 10 kHz, a high order Finite Impulse Response (FIR) bandpass filter was designed in MATLAB using the signal processing toolbox and is shown in Figure 4.6.

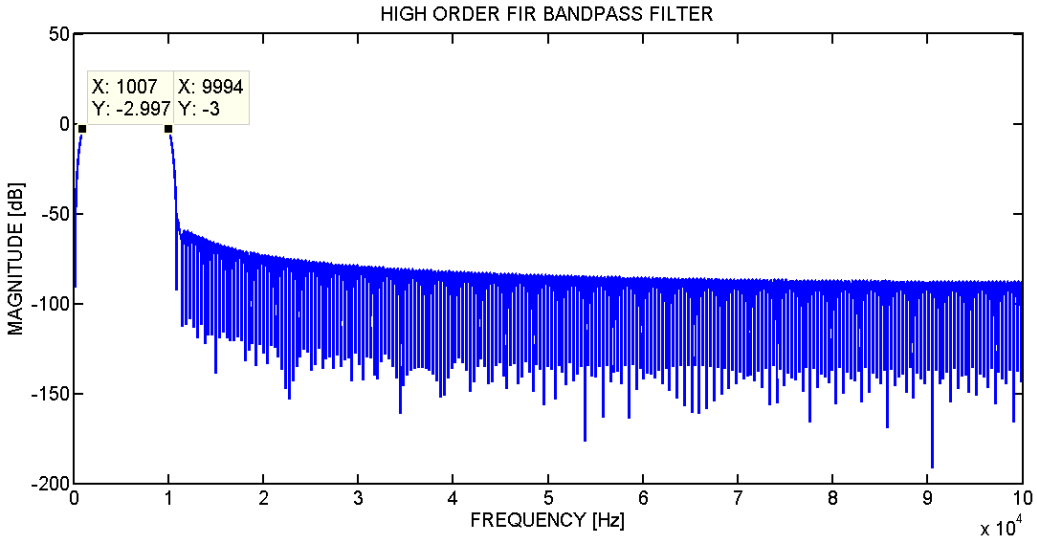

Figure 4.6: High order FIR bandpass filter used to filter data in Figure 4.4

The data in Figure 4.4 is then bandpass filtered and shown in Figure 4.7. This signal with minor overall gain adjustments will be used as a simulated noise source in the beamformer implementations of section 4.3 and 4.4.

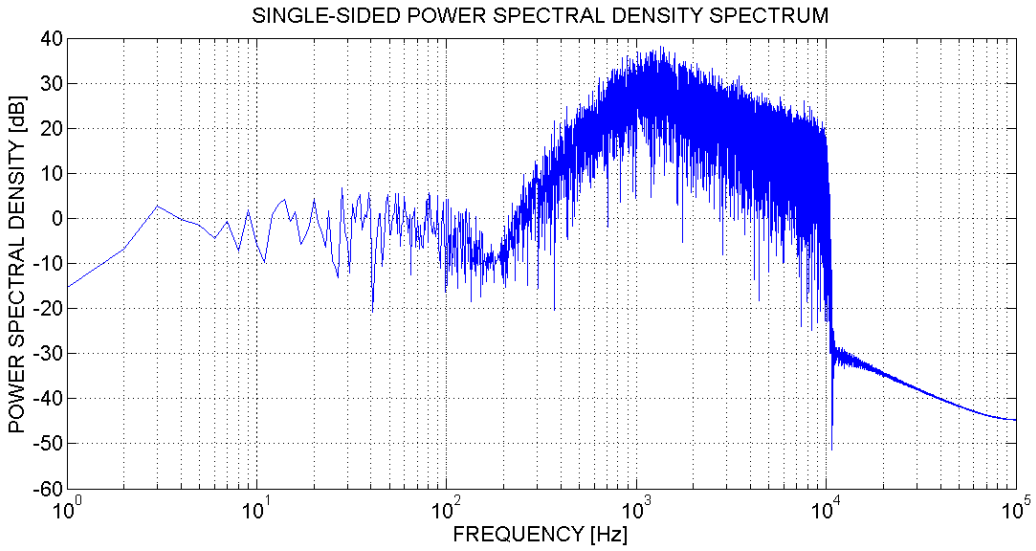

Figure 4.7: Data in Figure 4.4 bandwidth filtered from 1 to 10 kHz with the filter in Figure 4.6

4.2.2 Ambient Noise

Background noise limits the performance of a passive sonar system (Refer to section 2.7). It consists of ambient noise, reverberation, and self-noise of the sonar platform [5]. In this thesis, simulations of background noise will include ideal ambient noise only. Ideal ambient noise is often assumed in simulations and is of an isotropic and homogeneous nature [5].

The sources of ambient noise are varied, but mainly consist of pressure waves resulting from tide and sea waves and pressure impulses from sea turbulence [5]. Other contributors include rain, seismic activity, wind, molecular thermal motion, and fish schools [5].

Figure 4.8 shows the mean isotropic spectrum levels of ambient noise with different sea states. The sea state is a measure of how calm the sea is and takes into account the wave height and wind speed [9].

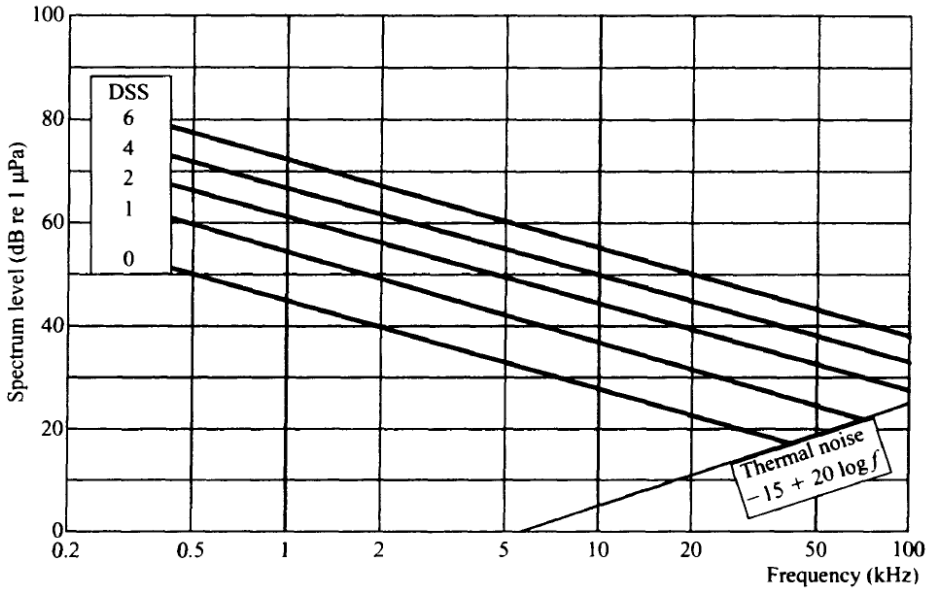

Figure 4.8: Ambient noise spectra for different sea states [9]

The simulation of ambient noise was done with the same algorithm and procedure for the source as in section 4.2.1 with one key difference: in the band of 1 to 10 kHz, ambient noise has a spectrum decay of -17 dB instead of -20 dB per decade.

See Figure 4.9, Figure 4.10, Figure 4.11, and Figure 4.12 for the approximated IIR shaping filter, single-sided power density spectrum of the filtered data, average single-sided power density spectrum, and the final ambient noise signal filtered to the band of 1 to 10 kHz respectively.

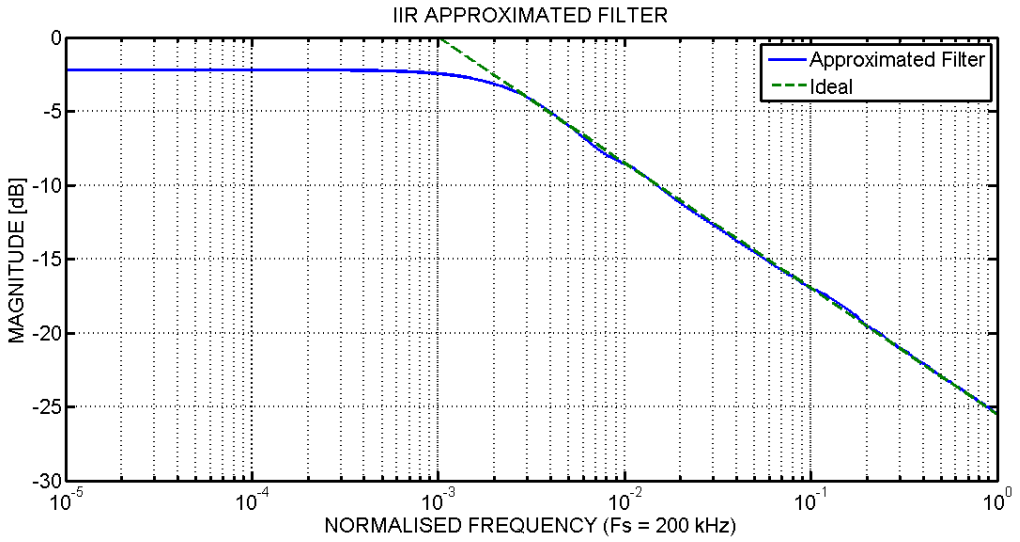

Figure 4.9: Frequency response of the approximated IIR filter for ambient noise simulation

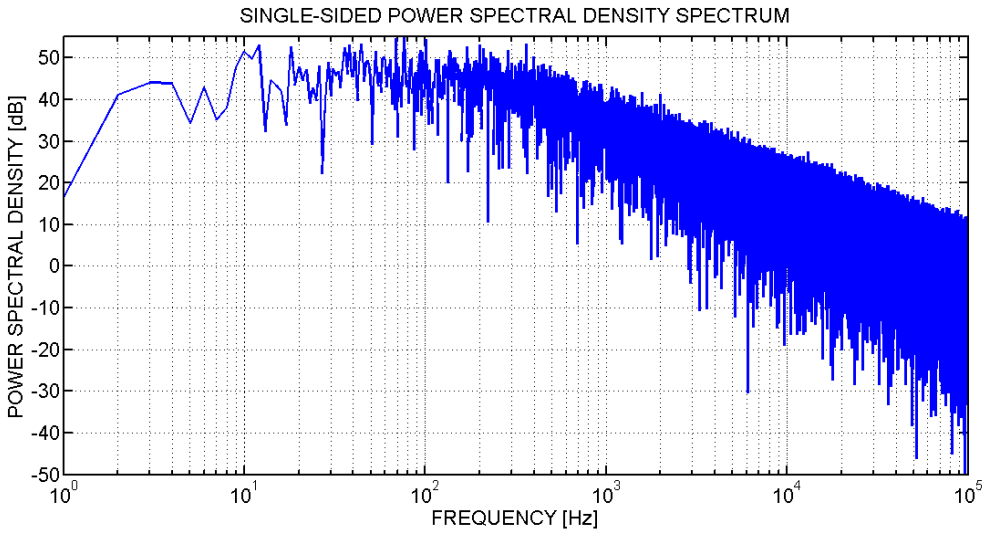

Figure 4.10: Single-sided power spectral density spectrum of filtered Gaussian white noise for ambient noise simulation

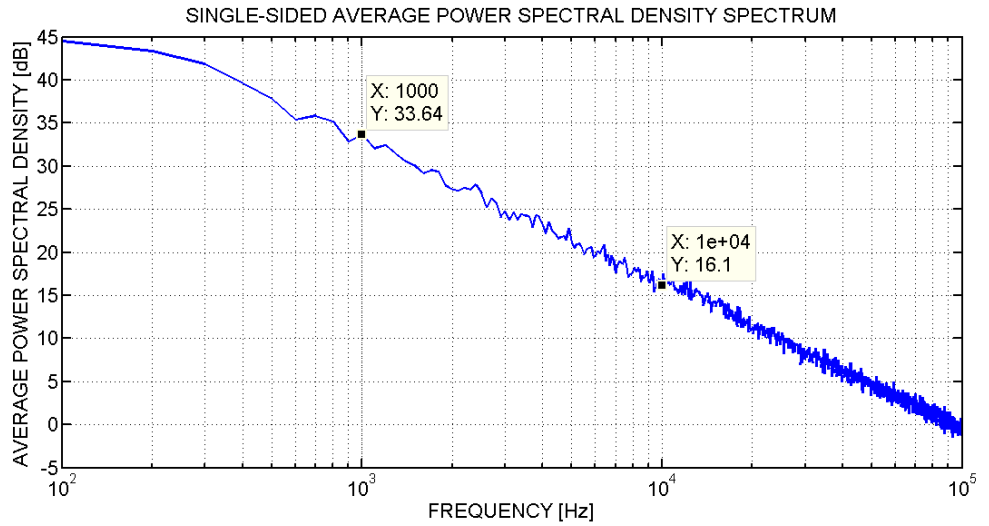

Figure 4.11: Single-sided average power spectral density spectrum of Figure 4.10

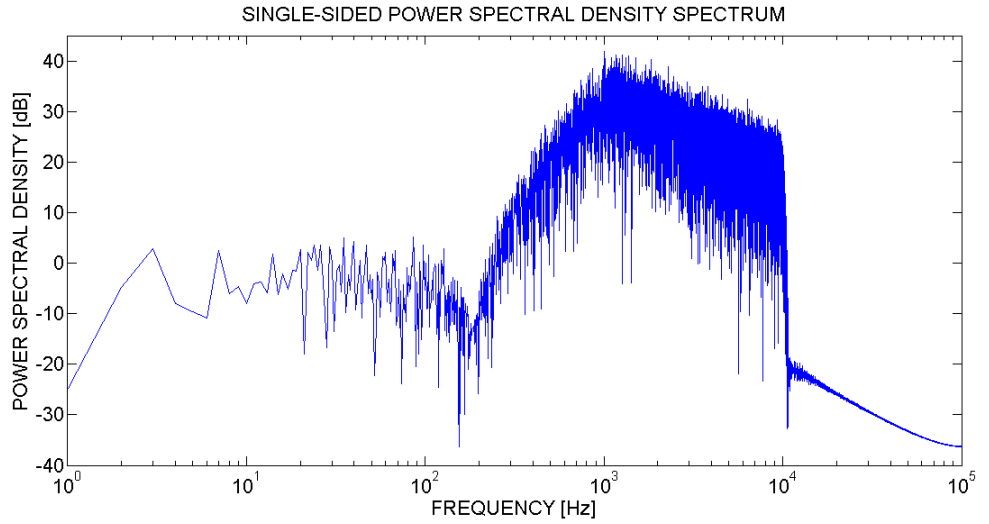

Figure 4.12: Data in Figure 4.10 bandwidth filtered from 1 to 10 kHz with the filter in Figure 4.6

4.2.3 Passive Sonar Signal for Use in Beamformer Simulations

The signal that will be used in the beamformer simulations of section 4.3 and 4.4 is a combination of the simulated source and ambient noise signals in the preceding sections.

It was mentioned in section 4.2.2 that the ambient noise is assumed to be isotropic and homogeneous. A further common assumption applied in these beamformer simulations is that the noise at each sensor is independent of one another [5].

The Gaussian signals of the source and ambient noise are normalised to have a mean of zero and a variance of one. To achieve the desired signal to noise ratio, the source is multiplied by a gain factor before the signal and the ambient noise are summed to represent the signal that was captured by a sensor. To achieve the desired SNR between the two signals, the equation for the SNR is solved for the gain required. The SNR for two signals, with power P_N and P_S for the noise and source respectively, is defined as [3]:

$$SNR = \frac{P_S}{P_N} \tag{4.2.1}$$

To calculate the power of a set of samples, the RMS is squared. Assuming we have a constant gain α multiplied with the source signal samples x_s before the RMS is calculated, the RMS is defined as:

$$RMS = \alpha \times \sqrt{\frac{1}{N}\sum_{n=1}^{N} x_s^2\,[n]} \tag{4.2.2}$$

With this equation for RMS, the SNR then becomes:

$$SNR = \frac{\alpha^2 P_S}{P_N} \tag{4.2.3}$$

Solving for the gain α:

$$\alpha = \sqrt{\frac{SNR \times P_N}{P_S}} \tag{4.2.4}$$

This gain is multiplied with the source signal before the source and ambient noise signals are summed for each sensor to achieve the desired SNR at each sensor. It is worth noting that when the mean of both signals are zero, the variance of the signal is equal to the RMS [5]. Thus, with the normalised signals of zero mean and a variance of one, equation 4.2.4 in this case becomes $\alpha = \sqrt{SNR}$.

Each sensor in the array of the beamformer simulations will receive a simulated signal that is a combination of simulated source and ambient noise with a specific SNR as mentioned above. For each sensor, the signal is firstly shifted in time with the appropriate amount of samples

corresponding to the chosen position of the source. This time shift of the signal is essentially the reverse of the delay-and-sum beamforming algorithm to simulate the captured signals of the sensors as if it was done in practice.

4.3 Delay-and-Sum Beamformer Simulations in MATLAB

This section discusses delay-and-sum beamforming simulations done in MATLAB. As mentioned in section 2.4.1, this thesis will focus on digital delay-and-sum beamforming. Firstly, the delay matrix used for calculating the appropriate delay, given the sensor positions and possible source positions, will be explained. Following this is a discussion of delay-and-sum beamforming simulations for a two-dimensional- and three-dimensional array, both where the source is in the near-field or in the far-field.

4.3.1 Delay Matrix

The delay matrix for each sensor is used to know the delay in samples that each of the sensor's received signals in the array needs to be delayed with to form a delay-and-sum beam. A beam can be formed in a direction (planar wave – far-field beamforming) or at a position (circular wave – near-field beamforming).

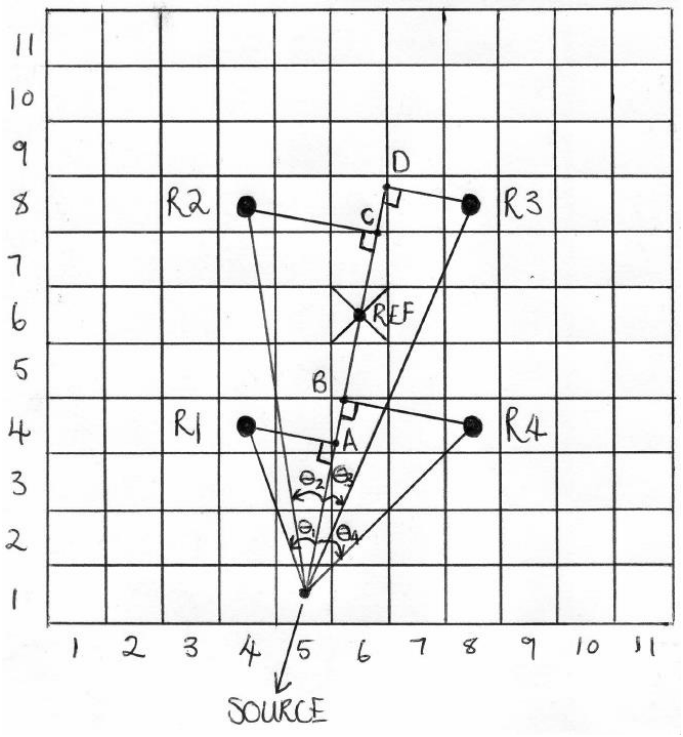

Figure 4.13: A grid in two-dimensional space containing a rectangular array with four sensors R1, R2, R3, and R4

Figure 4.13 shows a 2D grid in space with a source at (5, 1) and sensors R1, R2, R3, and R4 at (4, 4), (4, 8), (8, 8), and (8, 4) respectively. Also shown is a reference point REF at (6, 6) and its purpose will be explained below. Each block's size in the grid represents an arbitrary distance in metres.

Each sensor's time delay matrix will have an entry at each of the blocks in the grid and thus have the same size as the grid. The following is needed to calculate the sample delay matrix for each sensor:

1. The appropriate distance (in blocks) that a circular or a planar wave needs to travel from the assumed position of the source to a sensor in the array. Thus, it should be known if near-field or far-field beamforming should be applied as the distance that these waves need to travel from the source to a sensor is different for these two types of beamforming.
   In the near-field case the distance from the source to a sensor that the wave has to travel is the shortest distance from the source to the sensor (E.g. the distance in blocks of the line from the source at (5, 1) to R1 at (4, 4) in Figure 4.13). This distance can be calculated using Pythagoras' theorem.
   For the far-field case the distance from the source to sensor R1 that a planar wave has to travel will be the distance from the source at (5, 1) to point A marked on the line that goes from the source through the middle of the array (marked with REF) in Figure 4.13. This distance can be calculated using the law of cosines.
2. The appropriate distance (in blocks) that a circular or a planar wave needs to travel from the assumed position of the source to the reference point (middle) of the array.
3. The temporal sampling frequency of the received signals at each sensor.
4. The size in metres that each block's side represents.
5. The speed in metres per second at which a signal wave travels through the medium in which it propagates. The speed of signal waves in seawater is usually determined as a function of temperature, salinity, and depth. Alternatively, it can be measured directly [3]. Usually it is assumed to be 1500 m/s when the abovementioned factors are not taken into account [3].
6. The chosen position of the array's reference point (marked with REF in the figure above).

When all the variables mentioned above are known, the sample delay matrix for each sensor can be calculated with all the possible source positions in the grid taken into account.

Denoting the distance in blocks from a point in the 2D grid (x, y) for the m'th sensor as $D_m(x, y)$, the size in metres of the side of each block in the grid as G_{size}, the temporal sampling rate in samples per second as F_s, and the speed in metres per second at which the wave travels V_{signal} then the delay in samples n_m necessary at the position of (x, y) for the m'th sensor, is given by:

$$n_m(x, y) = \frac{D_m(x, y) \times G_{size} \times F_s}{V_{signal}} \quad [samples] \qquad \textbf{4.3.1}$$

The delay in samples n_m thus represents the amount of samples that a wave needs to "travel" before it reaches the m'th sensor. A unit simplification of equation 4.3.1 is shown below.

$$n_m(x,y) = \frac{D_m(x,y) \times G_{size} \times F_s}{V_{signal}}$$
$$= \frac{1 \times metres \times \frac{samples}{second}}{\frac{metres}{second}}$$
$$= samples$$

To get the final sample delay matrix that will be used for delay-and-sum beamforming, the sample delay matrix is also calculated for the reference point of the array and subtracted from the sample delay matrix of each sensor according to:

$$n_{final}(x,y) = n_m(x,y) - n_{REF}(x,y) \qquad \textbf{4.3.2}$$

A circular shifting register is used to delay each of the sensor's received signal with the appropriate amount of samples. The reference sample delay matrix is subtracted from each sensor's sample delay matrix in order to avoid unnecessary amounts of shifting in the register. When a reference point is not used and the assumed source position is far away from the array, the amount of shifts in samples will be very high and is in fact unnecessary because the sensor only receives the signal when it arrives at the sensor. Thus, to achieve constructive interference of a source at a certain position or in a certain direction the only delays that needs to be taken into account is the amount of samples it took for the signal to propagate between the sensors of the array and using a reference point in the array achieves this.

Consider the 2D rectangular array in an 81 by 81 blocks grid in Figure 4.14 where each sensor is marked with a blue cross. As mentioned above, each block represents an arbitrary distance in metres and in this case, the array has a distance of two blocks between each sensor. The reference point used for this array is in the middle of the array at (41, 41).

Calculating the far-field beamforming delay matrix in blocks without a reference point for the sensor in the top left corner and plotting it yields Figure 4.15 and Figure 4.16 with the side and top view respectively. These figures represents the plot of $D_m(x,y)$ in equation 4.3.1. As previously discussed, far-field beamforming can only determine the bearing of the source signal. Thus, it is only necessary to calculate the delay matrix for a discrete sphere of a certain resolution (in blocks) around the array where the delay matrix in this case is calculated for each block in the grid for illustrative purposes.

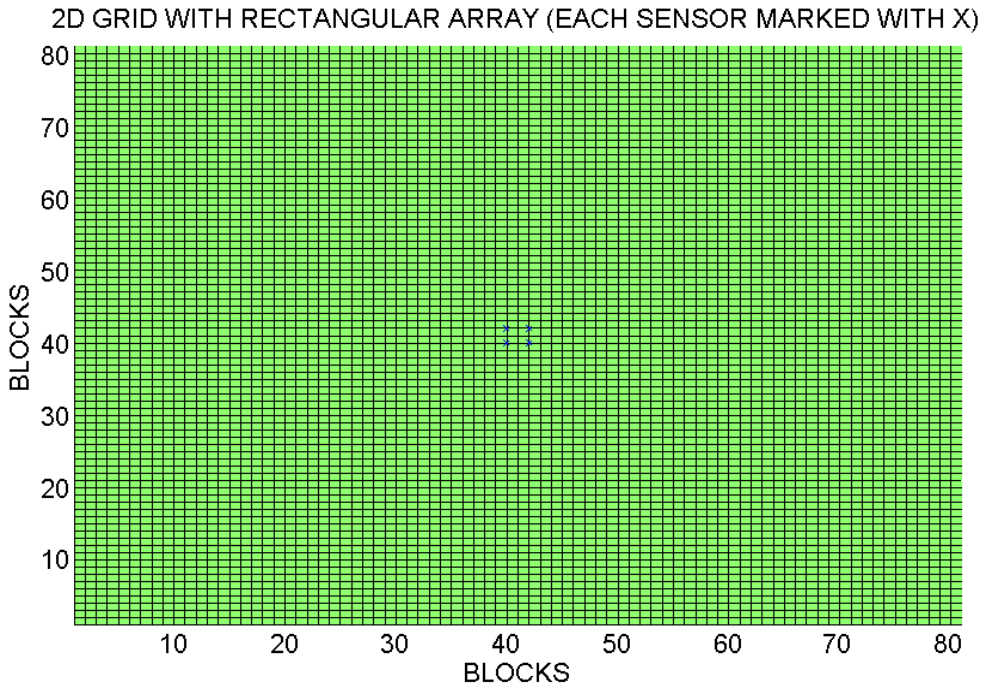

Figure 4.14: Two-dimensional grid with a four sensor rectangular array in the middle where each sensor is marked with a blue cross.

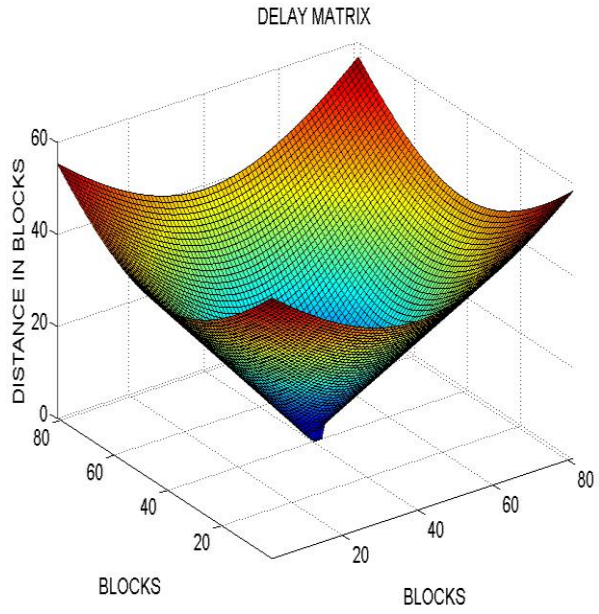

Figure 4.15: Side view of the far-field delay matrix in blocks for one sensor of a rectangular array.

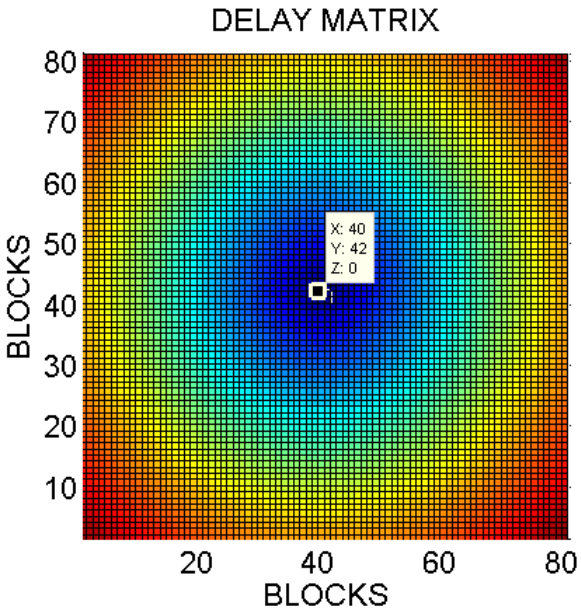

Figure 4.16: Top view of Figure 4.15.

Subtracting the reference point's delay matrix of the same kind at each block position according to equation 4.3.2 and plotting it yields Figure 4.17 and Figure 4.18 for the side and top view respectively. Notice that the delay in blocks at certain positions in the grid are negative. This effectively means that a circular shift to the left of the sensor data of the sensor is needed instead

of shifting it to the right as when positive delays are present. For the near-field beamforming case the plot will look similar but with different distances in blocks.

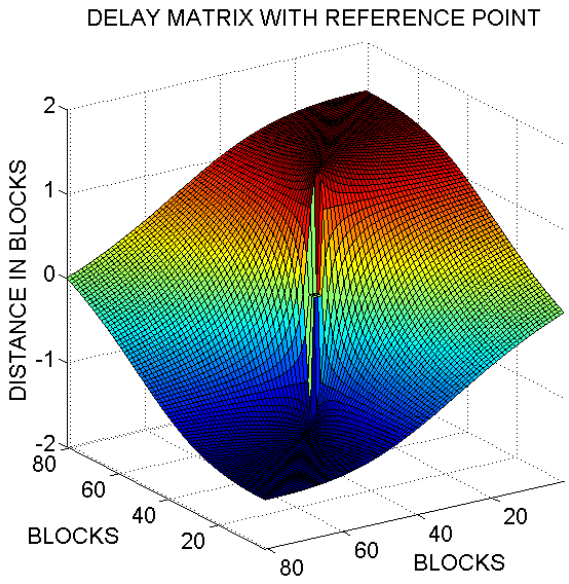

Figure 4.17: Side view of the far-field delay matrix in blocks, using a reference point, for one sensor of a rectangular array.

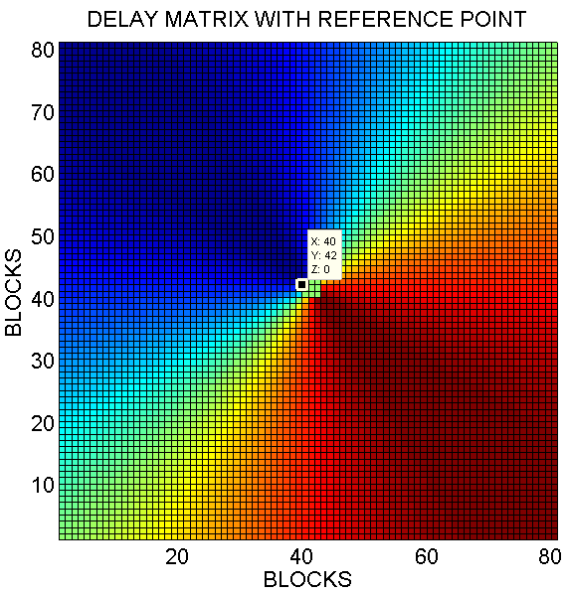

Figure 4.18: Top view of Figure 4.17.

To prepare the delay matrix for digital delay-and-sum beamforming, the delays in blocks present in $D_m(x, y)$ must be converted to delays in samples using equation 4.3.1. The maximum frequency of the simulated signals of section 4.2 is 10 kHz. The wavelength of 10 kHz is $\lambda = \frac{v}{f} = \frac{1500}{10000} = 0.15\ m$, resulting in a sensor spacing of 0.075 m. For the spacing of two blocks between sensors in the array discussed above, the size of each block thus needs to be $G_{size} = \frac{0.075}{2} = 0.0375\ m$. With $G_{size} = 0.0375\ m$, $F_s = 250\ kHz$, and $V_{signal} = 1500\ m/s$ in equation 4.3.1 the sample delay matrix ($n_{final}(x, y)$), rounded off to the nearest integer sample delay, for the same sensor used above is shown in Figure 4.19 and Figure 4.20 for the side and top view respectively.

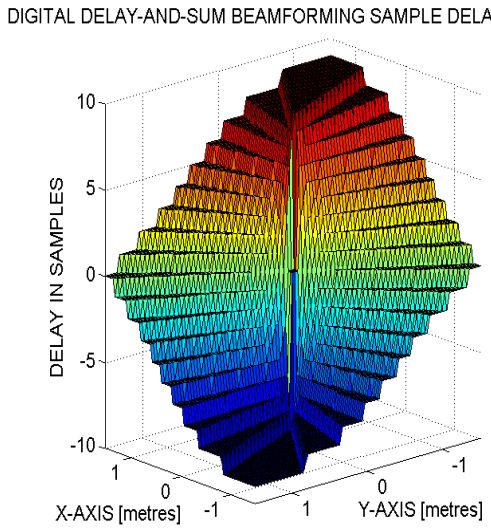

Figure 4.19: Side view of the far-field delay matrix in samples where $F_s = 250\ kHz$, using a reference point, for one sensor of a rectangular array.

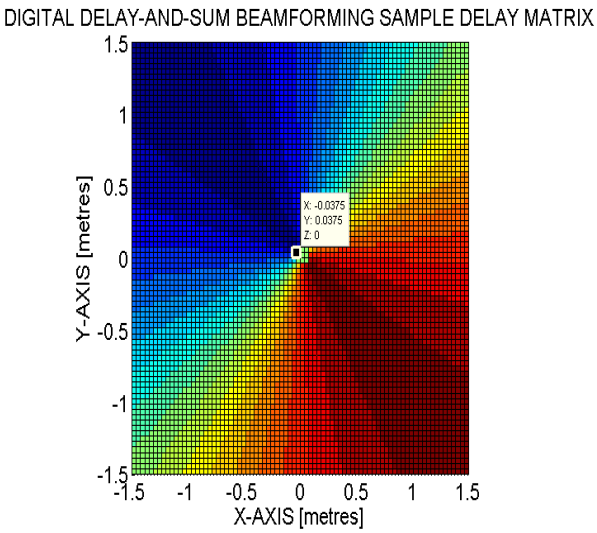

Figure 4.20: Top view of Figure 4.19.

Notice the coarseness of this delay matrix, introduced by rounding off to the nearest integer. To get better resolution in the delay matrix the factor $\frac{G_{size} \times F_s}{V_{signal}}$ in equation 4.3.1 needs to be maximised to increase the range of possible sample delays in the sample delay matrix. To avoid spatial aliasing, G_{size} needs to be equal to $\frac{\lambda}{2}$ and cannot be increased. V_{signal}, the speed at which a wave travels through the medium in which it propagates, is dictated by the medium itself. Thus the only term which can be changed to maximise this factor is F_s.

Increasing F_s from 250 kHz to 750 kHz gives a considerable increase in sample delay matrix resolution and is shown in Figure 4.21 and Figure 4.22 with a side and top view respectively.

The delays in samples in this sample delay matrix resembles more closely the actual delay in samples needed for a specific coordinate in the grid. As discussed in section 2.4.1, it is desirable to have a high temporal sampling frequency (higher than Nyquist sampling frequency) to yield more synchronous beams, yielding better resolution.

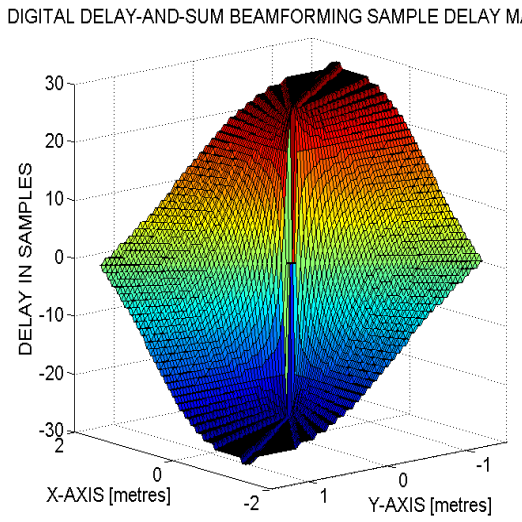

Figure 4.21: Side view of the far-field delay matrix in samples where $F_s = 750\ kHz$, using a reference point, for one sensor of a rectangular array.

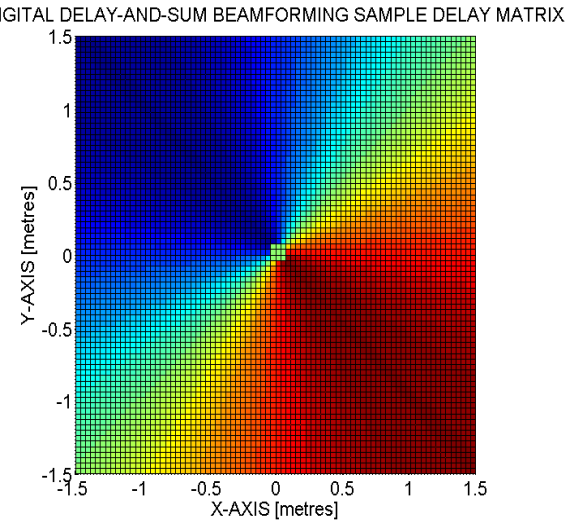

Figure 4.22: Top view of Figure 4.21.

4.3.2 Two-Dimensional Array

This section presents digital delay-and-sum beamforming applied to two-dimensional and three-dimensional arrays. The simulated signals of section 4.2 are used as input to the array. The specifications of the simulations are as follows unless otherwise stated:

- Frequency band of simulated signals for source and ambient noise: 1 to 10 kHz
- Temporal sampling frequency: $F_s = 200\ kHz$
- Speed of signal waves through medium: $V_{signal} = 1500\ m/s$
- Spacing between sensors (as calculated in section 4.3.1): $\frac{\lambda}{2} = \frac{0.15}{2} = 0.075\ m$
- With a spacing of two blocks between sensors, the size of each block in the grid needed to satisfy the spacing between sensors in metres is:

$$G_{size} = \frac{sensor\ spacing\ in\ metres}{sensor\ spacing\ in\ blocks} = \frac{0.075}{2} = 0.0375\ m$$

- Grid size: 81 by 81 $blocks$ = 81×0.0375 by 81×0.0375 $metres$ = 3.0375 by 3.0375 $metres$
- Signal to noise ratio: $SNR = 10log_{10}(0.5) = -3.0103\ dB$
- Array shading: Unshaded
- Grid resolution: 1:1 (1 beam formed per block in grid)

*Note that this set of specifications will serve as a basis to work from and will be referred to as the "**standard specifications**". All deviations from these specifications will be stated where applicable.*

4.3.2.1 Near-Field Beamforming – 4 Sensor Array

All results in this section are obtained using the four-sensor array as in Figure 4.14. Figure 4.23 shows the plot of a near-field delay-and-sum beamformer with standard specifications. The RMS power of each beam is calculated and plotted. The simulated source position is indicated with the data cursor. Note that the position of the source is not clear although it is one of 68 beams measured on the grid with maximum RMS power response.

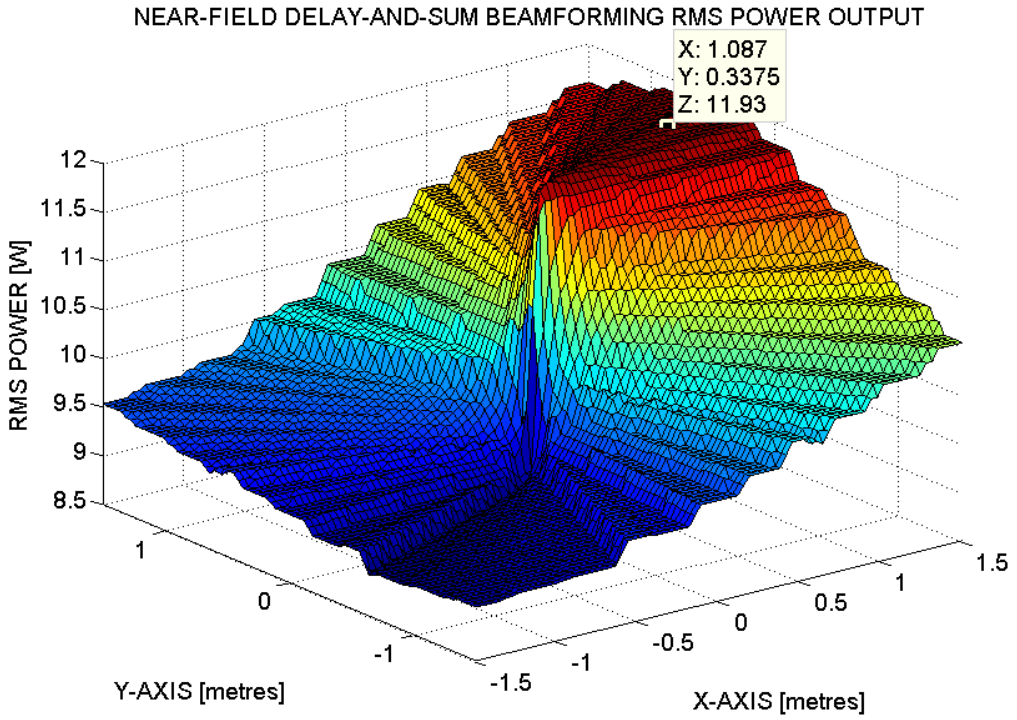

Figure 4.23: Near-field delay-and-sum beamforming RMS power plot with a 4-sensor array, standard specifications, and the source position indicated.

There are three factors causing the ambiguity of the source position, namely, the temporal sampling frequency F_s, the number of sensors in the array, and the resolution of the grid. The first two factors will increase the resolution of the beamformer through either increasing the number of synchronous beams by increasing F_s (as explained in section 4.3.1), or by increasing the array gain by increasing the number of sensors in the array. The last factor causes ambiguity through a grid resolution that is too fine (each block has a size of 0.0375 *m* by 0.0375 *m*) for the type of array and F_s used. The SNR of -3.01 dB also plays a small role but will be discussed in section 4.3.2.2. First, the effect of increasing F_s, decreasing grid resolution, and a combination of the two will be investigated respectively. Increasing the number of sensors in the array will be discussed in section 4.3.2.2.

Increasing F_s from 200 kHz (as in Figure 4.23) to 1.2 MHz yields Figure 4.24. The source is now one of only 8 positions measured on the grid with maximum RMS power response, decreasing from 68 positions previously.

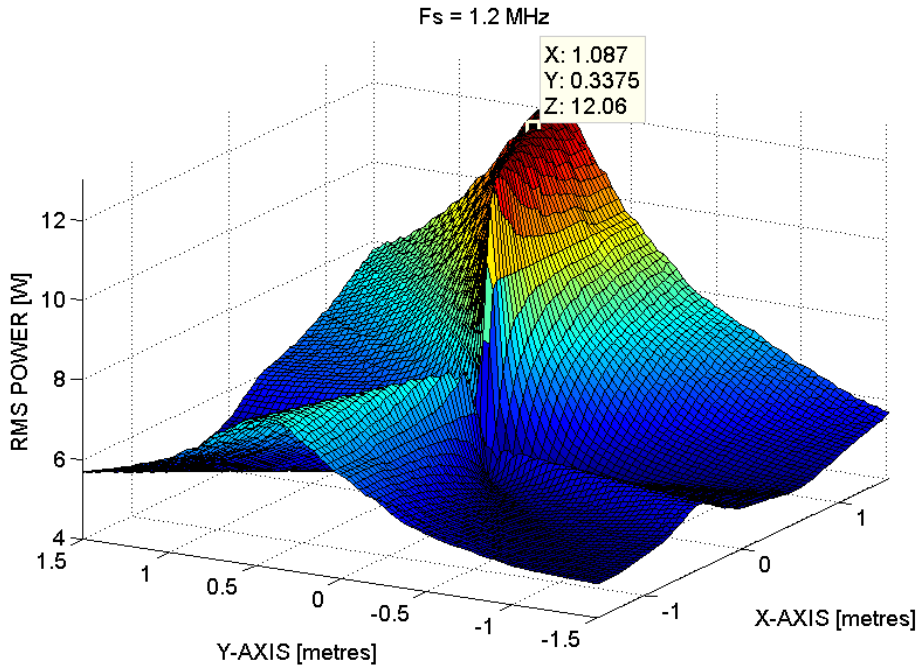

Figure 4.24: Near-field delay-and-sum beamforming RMS power plot with temporal sampling frequency increased to 1.2Mhz.

Decreasing the grid resolution of Figure 4.23 from 1:1 (1 beam formed per block) to 1:49 (1 beam formed per $7 \times 7 = 49$ blocks) yields Figure 4.25. Although neighbouring measurement blocks have RMS power close to the one containing the source (as indicated on the figure), maximum RMS power response is still achieved in the measurement block containing the source. In this case, decreasing the resolution from 1:1 to 1:49 is the minimum decrease in grid resolution needed to give maximum RMS power response in the block containing the source.

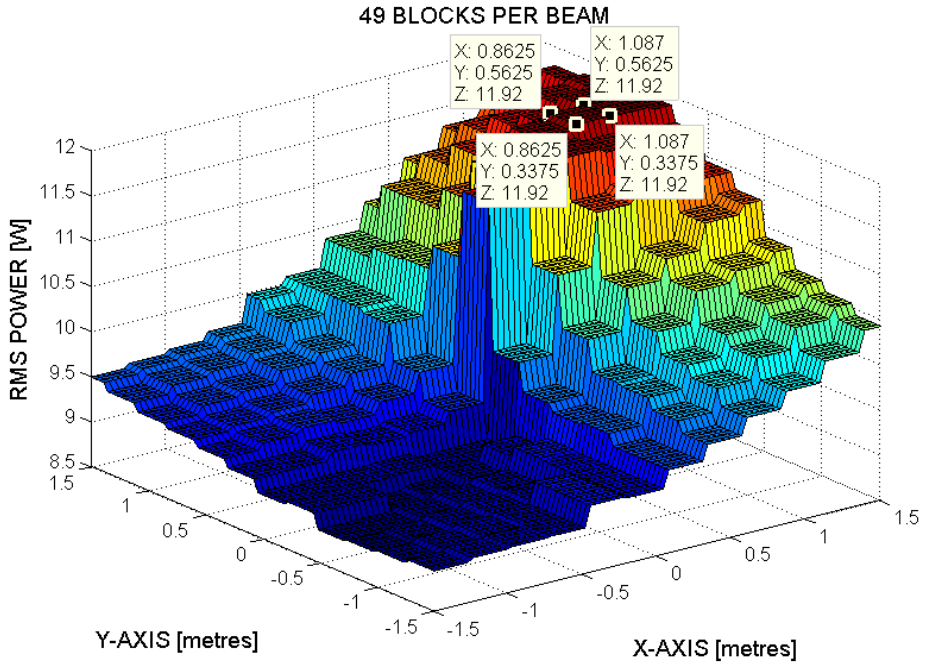

Figure 4.25: Near-field delay-and-sum beamforming RMS power plot with decreased grid resolution of 1:49 and with maximum RMS response area indicated containing the source position.

Taking the increased F_s case contained in Figure 4.24 and decreasing the grid resolution from 1:1 to 1:3 yields Figure 4.26. Maximum RMS power response is now contained at the source position as indicated in the figure. Again, decreasing the resolution from 1:1 to 1:3 is the minimum decrease in resolution needed to give maximum RMS power response in the block containing the source. As expected, after increasing the number of synchronous beams, the decrease in grid resolution needed to have the source in the maximum RMS power response block is now much less than with the case of $F_s = 200\ kHz$ in Figure 4.25.

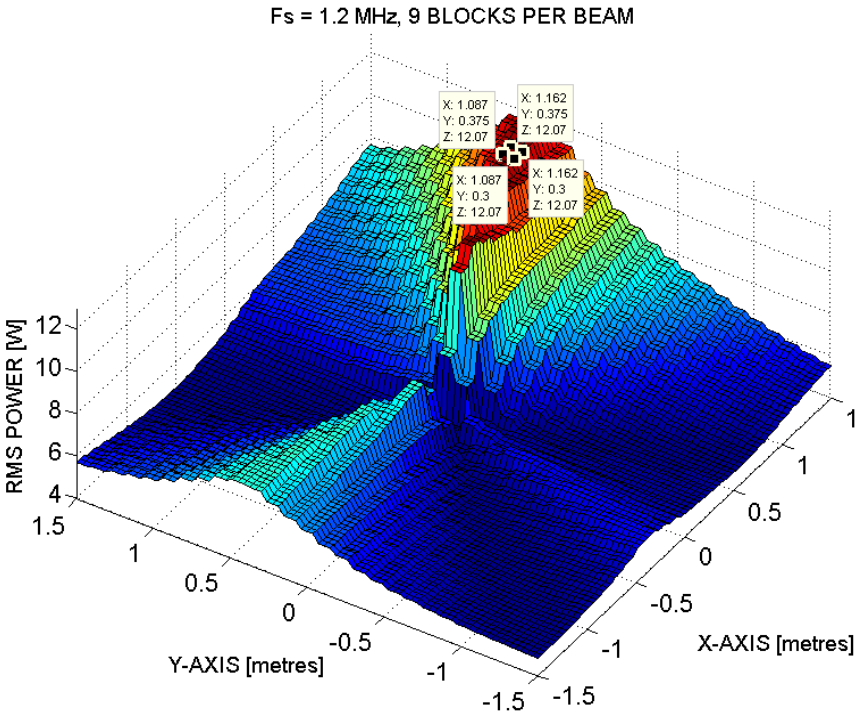

Figure 4.26: Near-field delay-and-sum beamforming RMS power with temporal sampling frequency increased to 1.2 MHz, decreased grid resolution of 1:9, and with maximum RMS power area indicated that contains the source position.

Additionally, decreasing the grid resolution translates to a decreased amount of beams that needs to be formed to scan the whole grid. This decreases computational power needed and facilitates scanning larger areas.

4.3.2.2 Near-Field Beamforming – 9 Sensor Array

Standard specifications contained in section 4.3.2 are also used in this section. Instead of a 4-sensor array as used in the previous section, a 9-sensor array as illustrated in Figure 4.27 will be used to produce the results in this section.

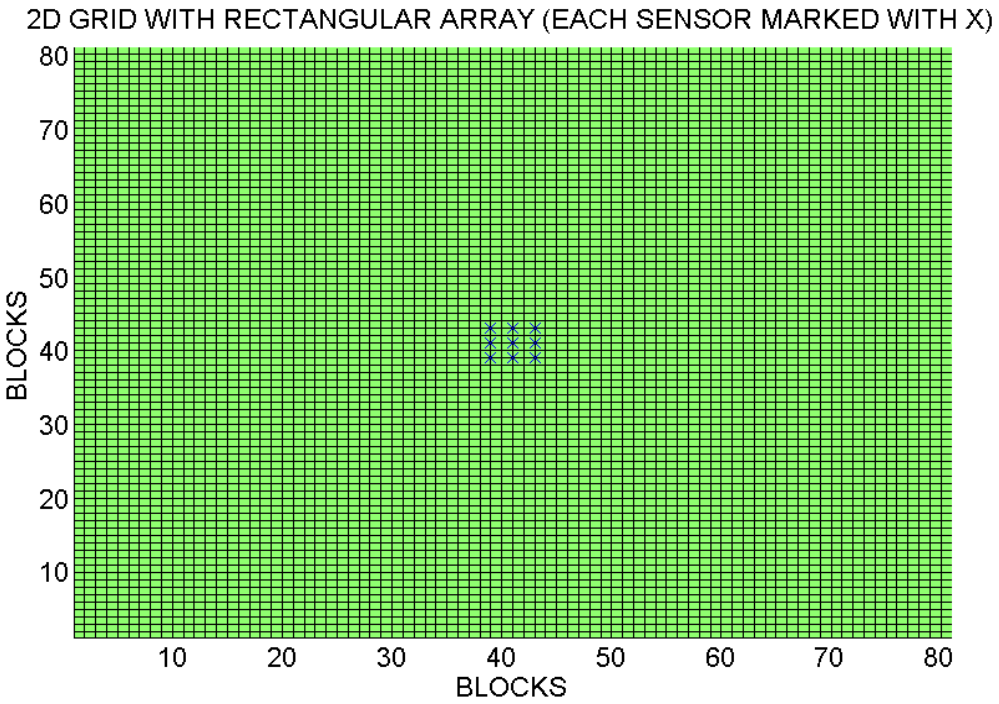

Figure 4.27: Two-dimensional grid with a 9-sensor rectangular array in the middle where each sensor is marked with a blue cross.

Figure 4.28 shows the delay-and-sum beamformer RMS power output with standard specifications. As expected, the output is smoother with the source position more pronounced when the array gain is increased by increasing the number of sensors. The source is now one of 14 maximum RMS power response beams, in comparison with the 68 maximum RMS power response beams as measured in the 4-sensor case.

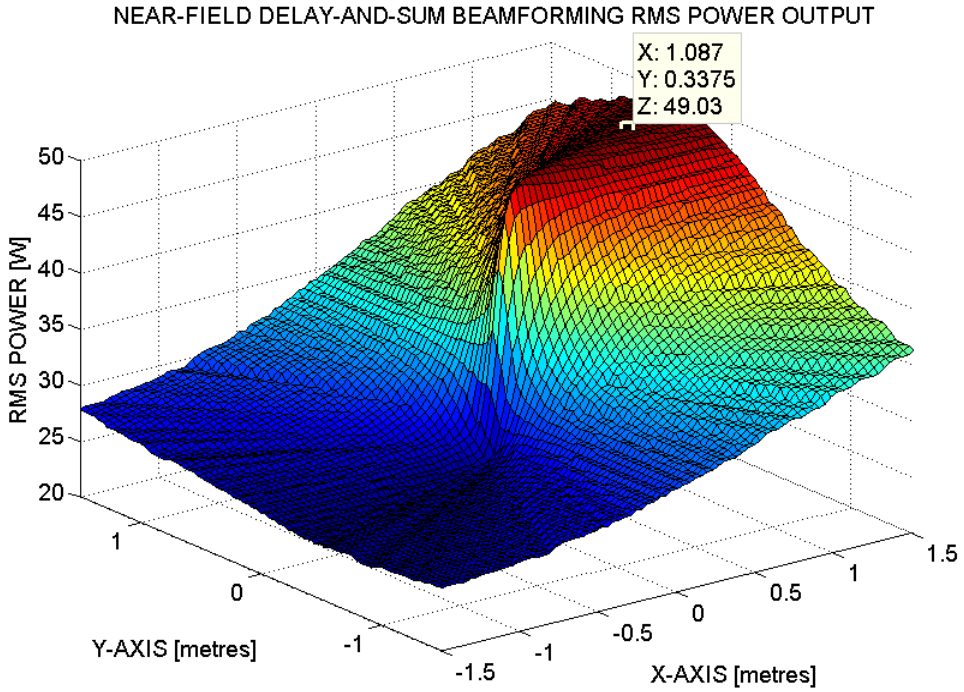

Figure 4.28: Near-field delay-and-sum beamforming RMS power plot with a 9-sensor array, standard specifications, and source position indicated.

The uppermost plot in Figure 4.29 shows the top view of Figure 4.28. The white circles and white dot indicates points on the plot where maximum RMS power response are achieved, where the dot is at the source position. Decreasing the grid resolution to 1:4 and 1:9 yields the other two plots in the figure. A grid resolution of 1:9 is the minimum decrease in grid resolution needed to achieve maximum RMS power response in the beam containing the source. As expected, this is significantly less than the decrease in grid resolution needed in the 4-sensor case (1:49) to achieve the same result.

Using standard specifications but increasing the temporal sampling frequency to 400, 600, and 800 kHz respectively are shown in Figure 4.30. The source position is the only maximum RMS power response beam in the plot when the temporal sampling frequency is equal to 800 kHz.

SNR also plays a large roll in beamformer performance. Using standard specifications but decreasing the SNR to -15.23, -20, and -30 dB respectively are shown in Figure 4.31. Already at -15.23 dB, the source position is not part of the maximum RMS power beams. The range of possible RMS power values for all the beams in the grid also reduces drastically with each step, making it increasingly difficult to distinguish between different beams. A SNR of -30 dB produces a near flat RMS power response in the grid, which is caused by the uncorrelated noises at each sensor.

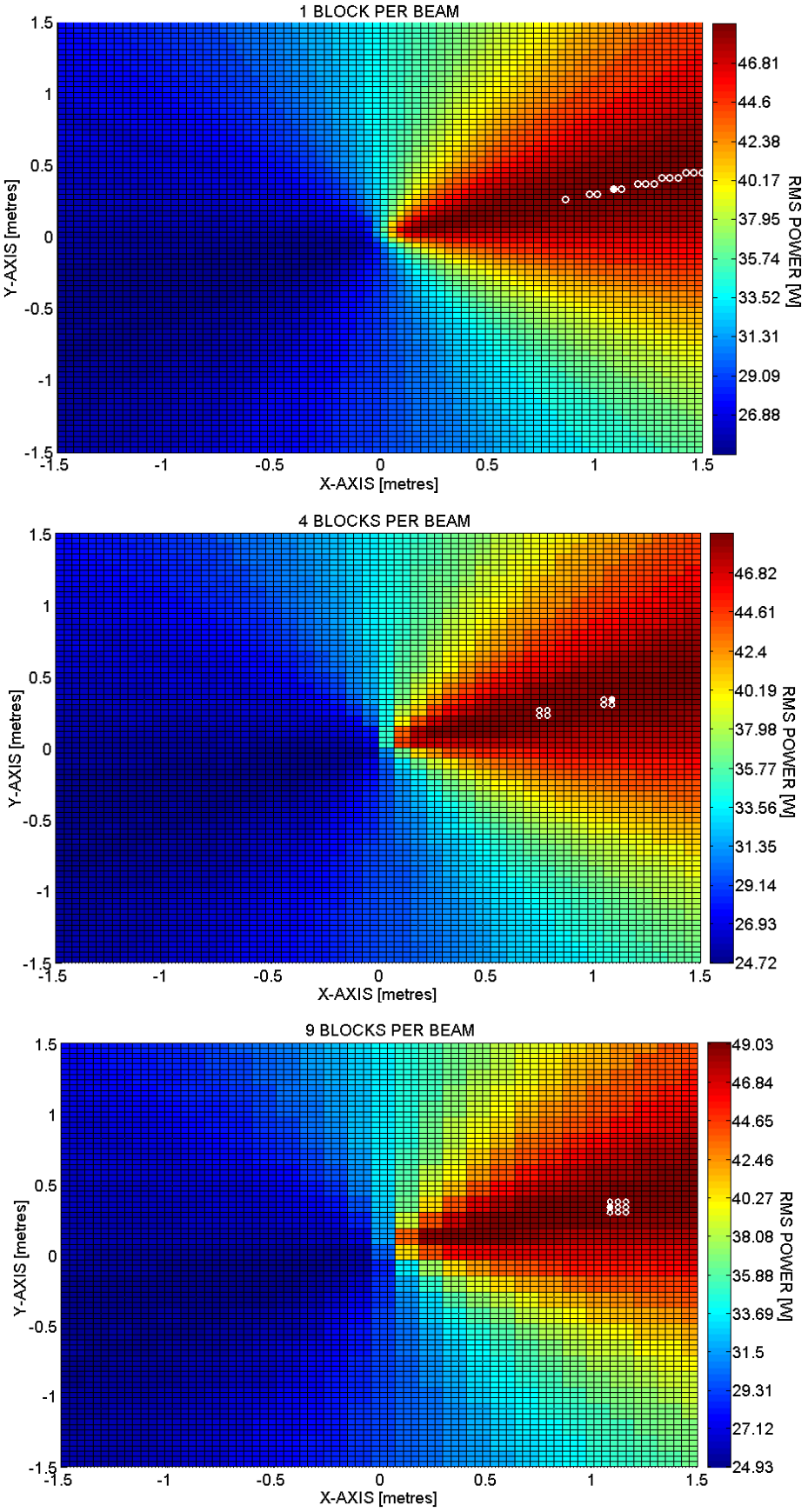

Figure 4.29: Delay-and-sum beamformer RMS power plots of gradually decreasing grid resolution. Maximum RMS power beams are indicated with white circles and a dot, where the dot is at the source position.

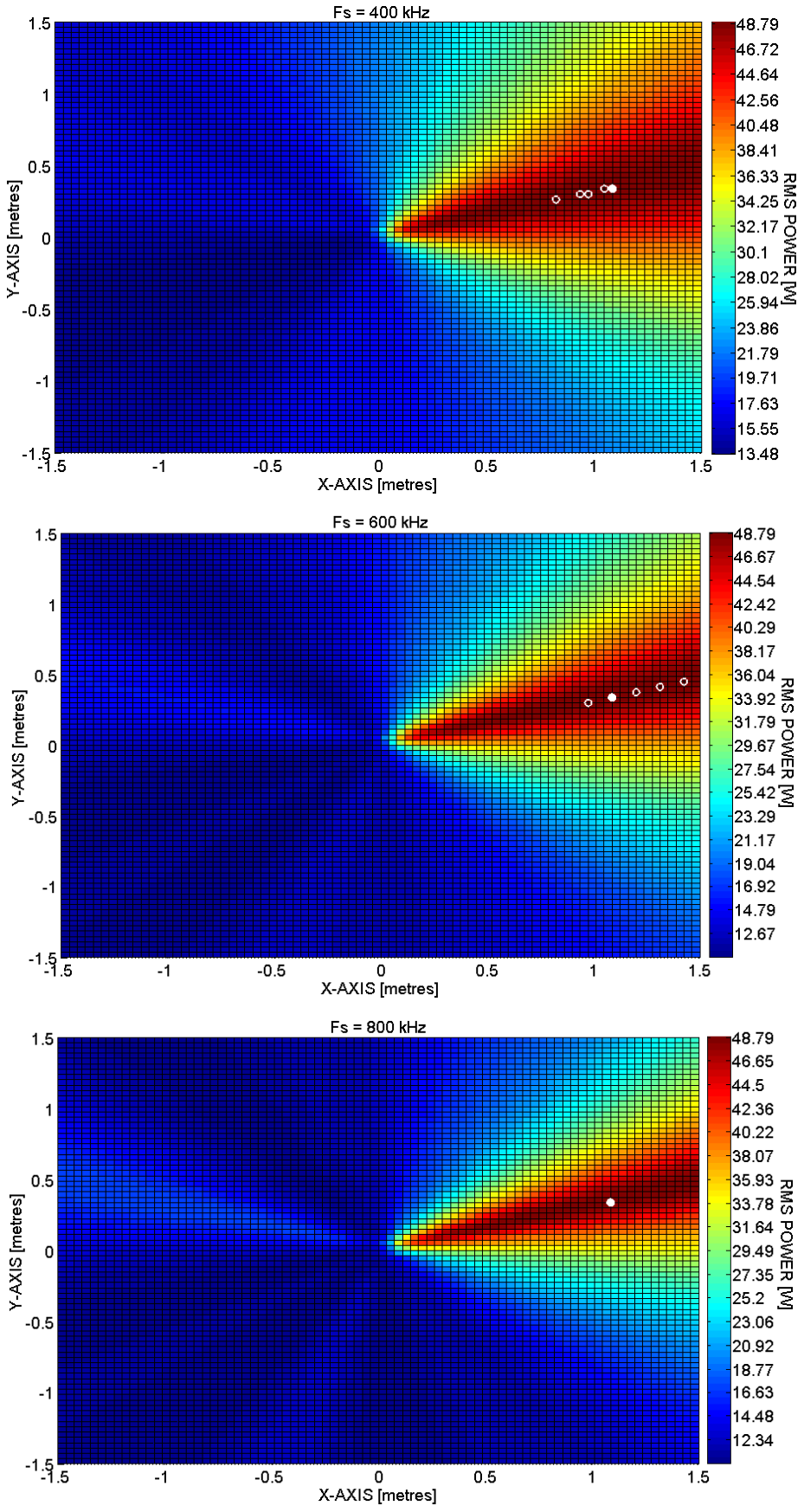

Figure 4.30: Delay-and-sum beamformer RMS power plots of increasing temporal sampling frequency. Maximum RMS power beams are indicated with white circles and a dot, where the dot indicates the source position.

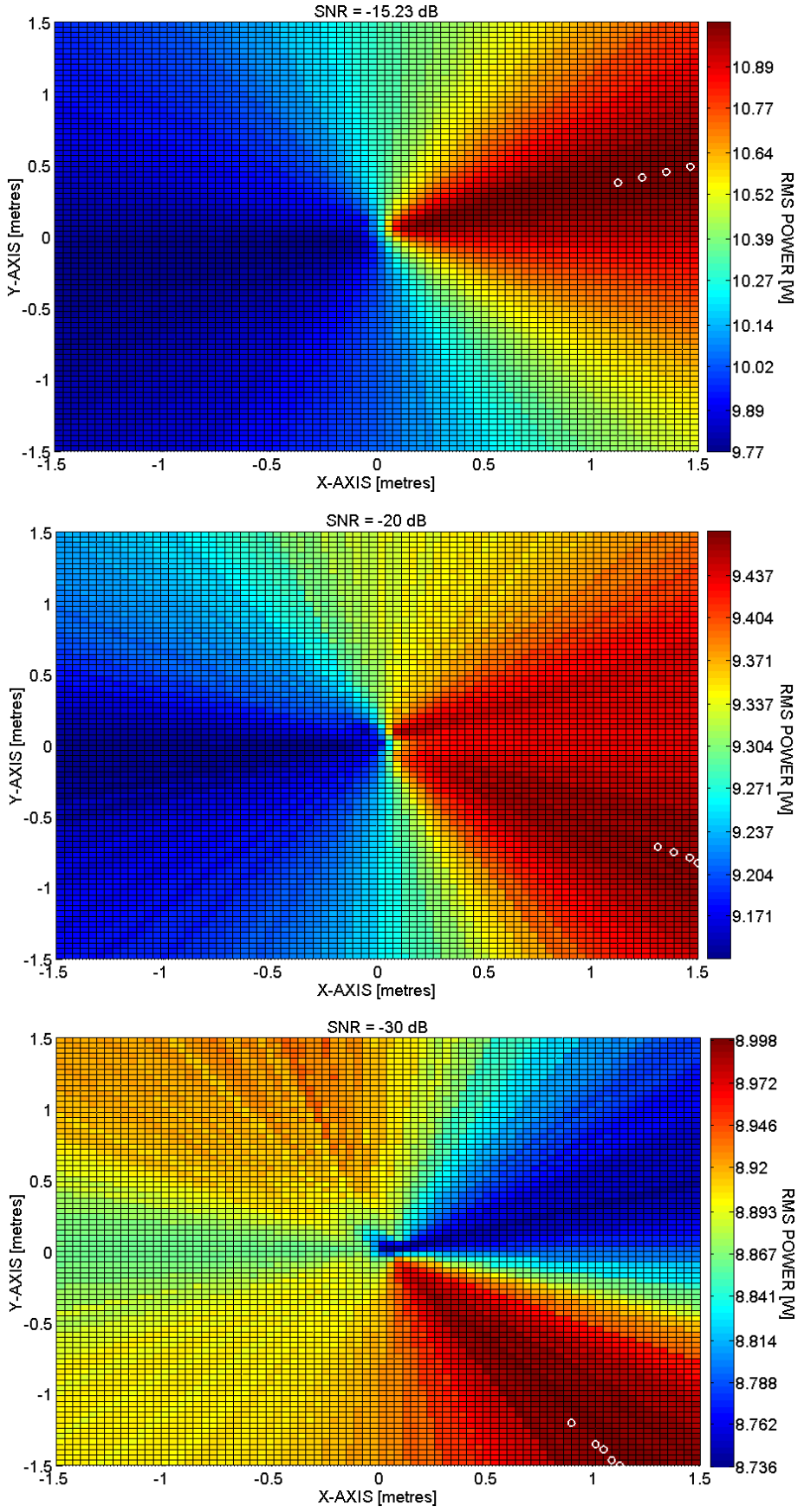

Figure 4.31: Delay-and-sum beamformer RMS power plots of decreasing SNR. Maximum RMS power beams are indicated with white circles and source position the same as in Figure 4.30.

Figure 4.32 shows the RMS power output of a delay-and-sum beamformer with temporal sampling frequency increased to 800 kHz, grid resolution decreased to 1:400, and the grid size significantly increased to 10 by 10 metres. Beams are only formed in the outer regions of the grid and the beam with maximum RMS power response is highlighted in white. With the source 10.33 metres away from the middle of the array, the source position is still identified under these conditions and marked with a black dot on the figure.

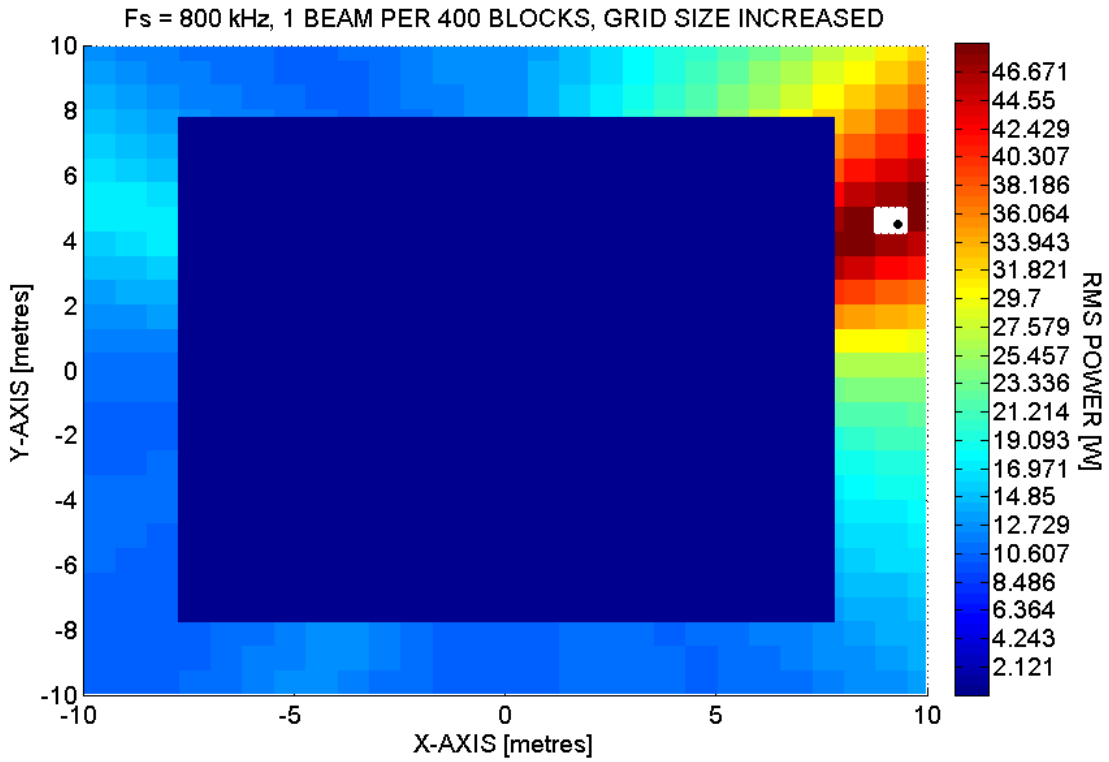

Figure 4.32: Delay-and-sum beamormer RMS power plot with temporal sampling frequency increased to 800 kHz, grid resolution decreased to 1:400, and grid size increased. Maximum RMS power response is indicated with the white block and the source position with the black dot.

4.3.2.3 Far-Field Beamforming – 9 Sensor Array

The same 9-sensor array that was used in the previous section will be used here (Figure 4.27). Because simulated signals are used, the signals received at each sensor can be prepared so that it represents a far-field source irrespective of distance.

When the source is far enough from a stationary array, or in the far-field, delay-and-sum beamforming can only distinguish bearing and not position. This effect can be seen in Figure 4.33 where the delay-and-sum beamformer RMS power outputs are shown for the near-field and far-field case with equal specifications. As with all previous simulations, a beam is formed at each position in the grid. Note that the near-field output has a finite amount of maximum RMS power beams on the line extending from the array in the middle of the grid through the source position to the outermost position of the grid. Conversely, each point on this same radial line has a maximum RMS power output in the far-field case. If this line was lengthened by extending the grid to the right, it will still have maximum RMS power output beams at each

position in the far-field case. However, no extra points will be introduced in the near-field case, given that the grid does not extend past the near-field distance of the array.

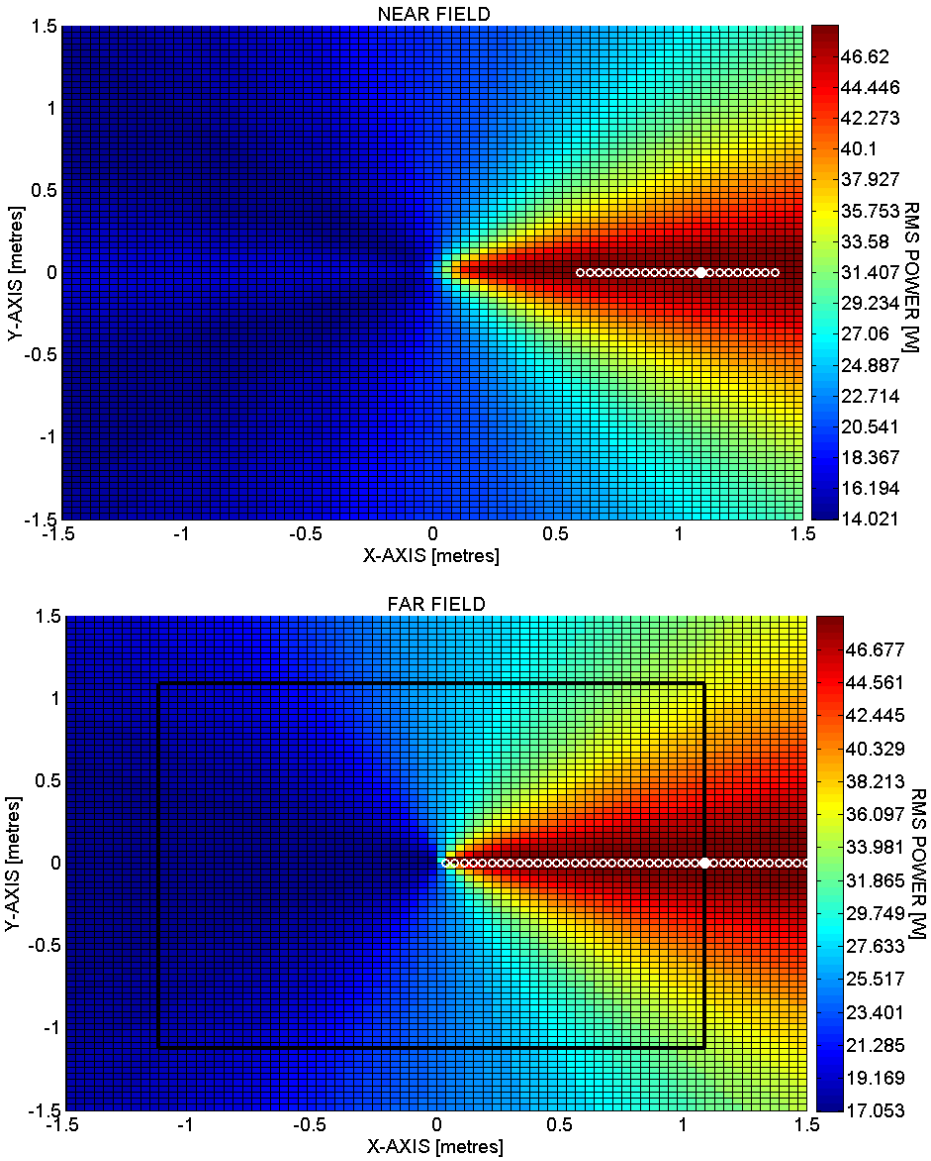

Figure 4.33: Far- and near-field delay-and-sum beamforming RMS power outputs with standard specifications except $F_s = 400\ kHz$. Maximum RMS power response beams are indicated with white circles and a dot, where the dot indicates the source position.

The remainder of the simulations done in this section will make use of a polar plot to represent the output of the far-field beamformer. The bearings at which the beams will be formed can be seen on the far-field output of Figure 4.33, marked with points that fall on the black rectangle. This approach is not optimal as the spacing between the different bearings at which the beams are formed are not equal but it is an effective approach nonetheless for output visualisation.

All of the phenomena when varying temporal sampling frequency, the number of sensors in the array, SNR, and grid resolution were explained in the preceding sections and will not be repeated here. Reducing grid resolution in the near-field beamforming case is equivalent to reducing the number of bearings at which beams formed in a 360-degree angle.

Figure 4.34, Figure 4.35, and Figure 4.36 shows the far-field delay-and-sum beamforming RMS power output for the standard specification, varying temporal sampling frequency, and varying SNR cases respectively.

In Figure 4.36, the uppermost and bottom RMS power output plots have the same SNR but varying temporal sampling frequency. Note that the higher temporal sampling frequency output is significantly more resilient to SNR degradation.

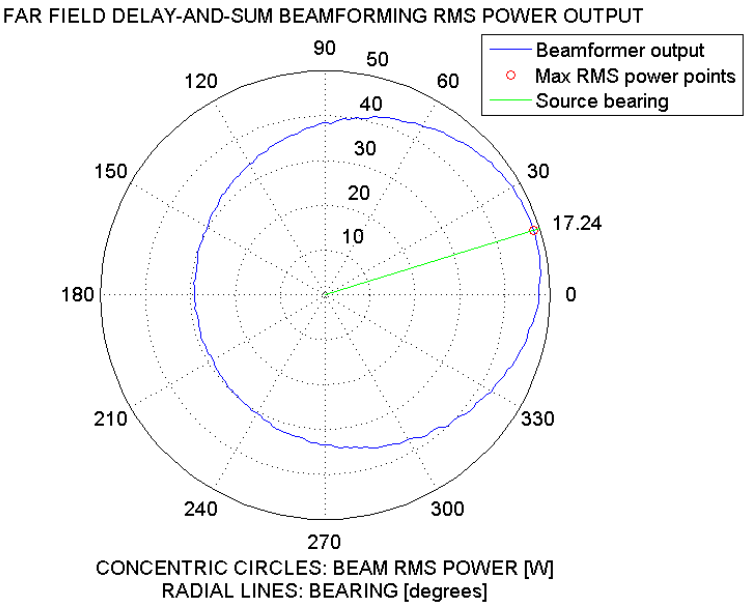

Figure 4.34: Standard specifications far-field delay-and-sum beamforming RMS power output polar plot of the data points falling on the black rectangle in Figure 4.33.

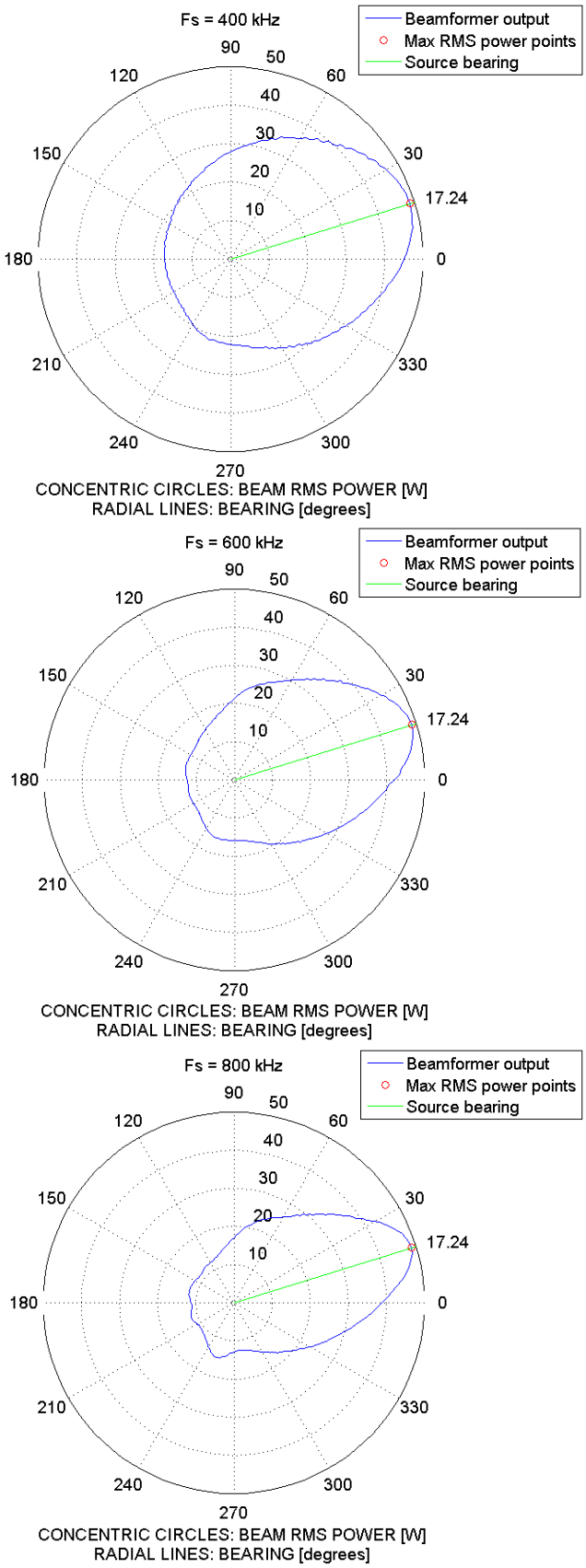

Figure 4.35: Far-field delay-and-sum beamforming RMS power outputs with increasing temporal sampling frequency.

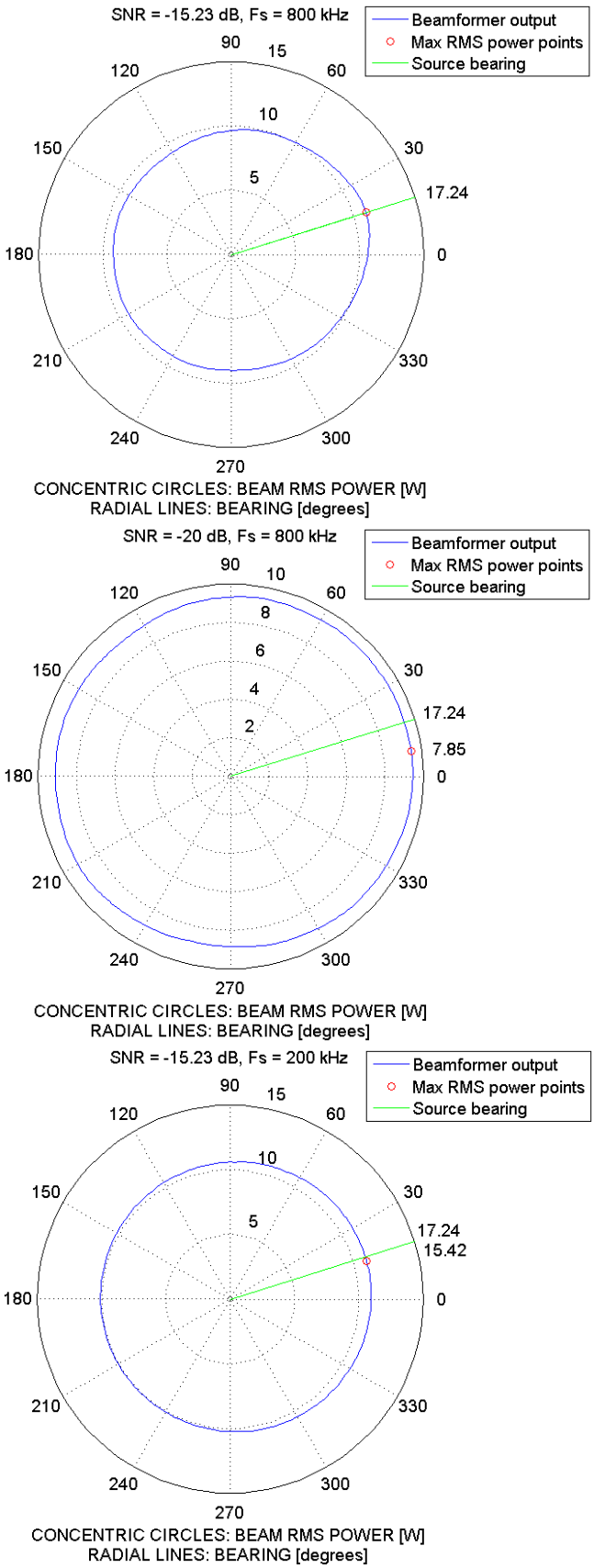

Figure 4.36: Far-field delay-and-sum beamforming RMS power outputs with varying SNR and temporal sampling frequency.

4.3.3 Three-Dimensional Array

The three-dimensional array used in this section is the same as the 4-sensor array in Figure 4.14 (section 4.3.1) but extended to three dimensions to form an 8-sensor cubic array with a sensor at each corner. Standard specifications will be used (unless otherwise stated) except that the grid size will be a 27×27×27 blocks cube.

Results of near- and far-field delay-and-sum beamforming will be presented in sections 4.3.3.1 and 4.3.3.2 respectively.

4.3.3.1 Near-Field

Figure 4.37, Figure 4.38, and Figure 4.39 shows the near-field delay-and-sum beamforming RMS power output for the standard specification, increased temporal sampling frequency, and varying SNR cases respectively. All observations with these variations are the same as in previous sections.

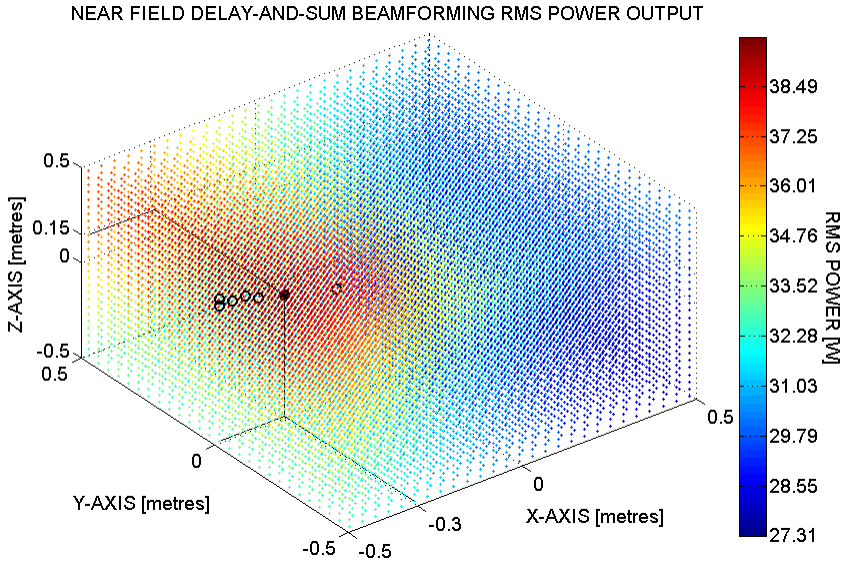

Figure 4.37: Near-field delay-and-sum beamforming RMS power output with standard specifications. Maximum RMS power response beams are indicated with black circles and a black dot, where the black dot represents the source position.

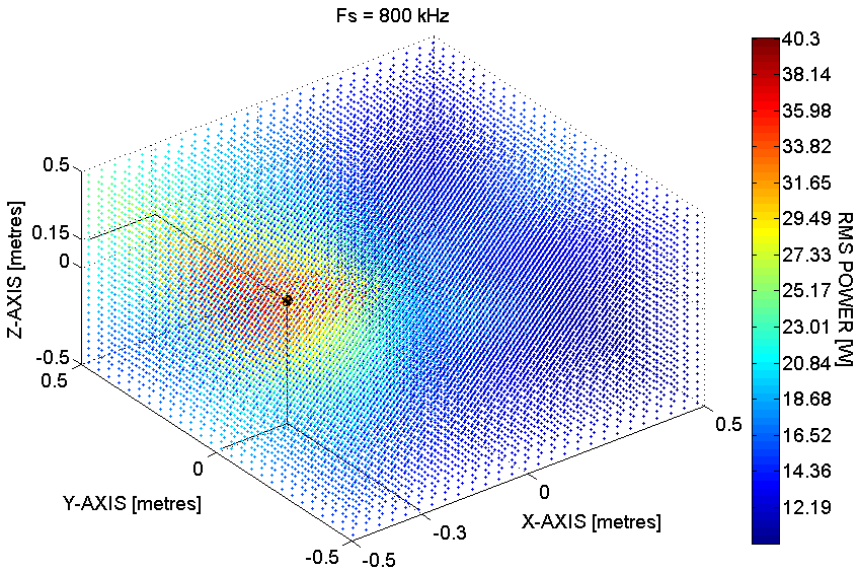

Figure 4.38: Near-field delay-and-sum beamforming RMS power output with increased temporal sampling frequency of 800 kHz. Maximum RMS power response beams are indicated with black circles and a black dot, where the black dot represents the source position. The source position is the only beam with maximum RMS power response.

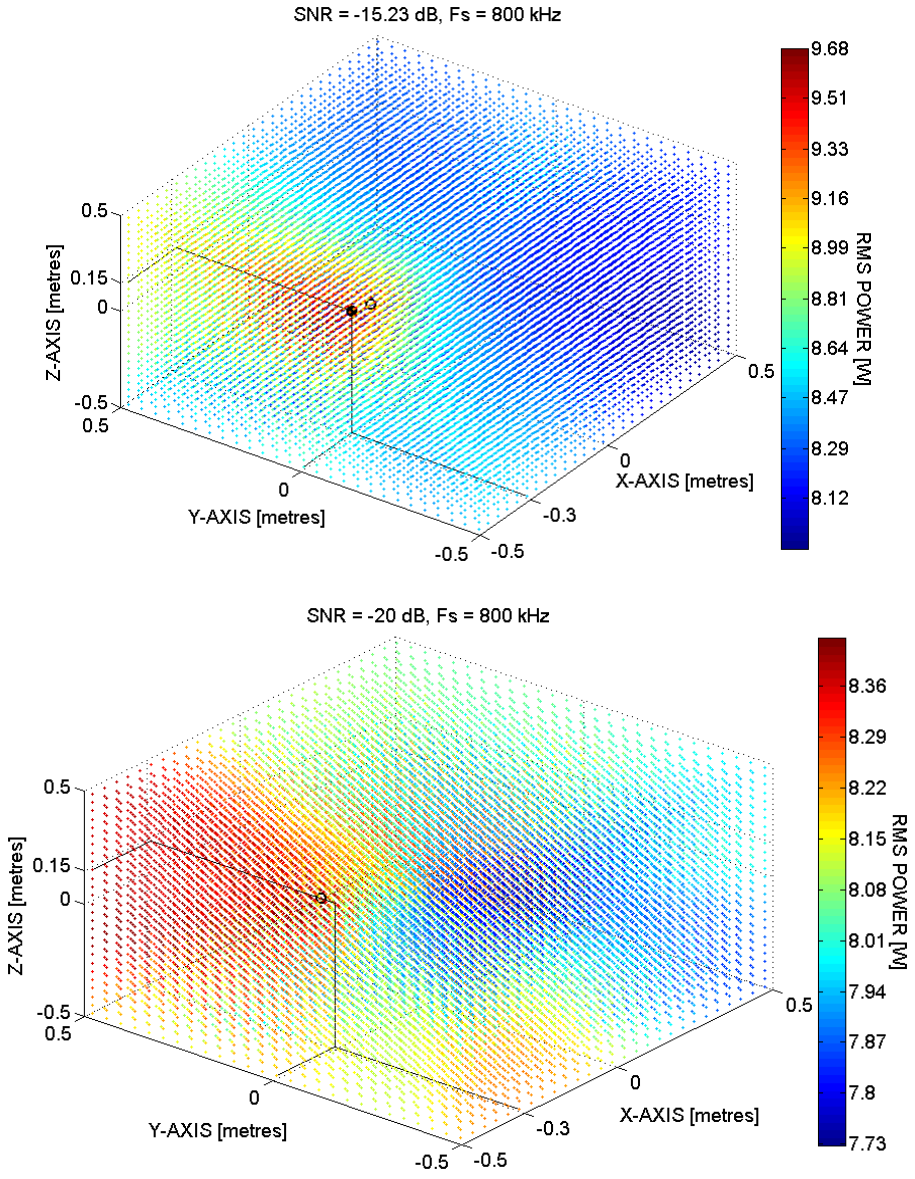

Figure 4.39: Near-field delay-and-sum beamforming RMS power outputs with varying SNR and temporal sampling frequency increased to 800 kHz. Maximum RMS power response beams are indicated with black circles and a black dot, where the black dot represents the source position.

4.3.3.2 Far-Field

The same method that was used in section 4.3.2 to obtain the bearings at which beams are formed in the 2D far-field case are used here. By using the points on the black rectangle in the bottom plot of Figure 4.33 and extending to three dimensions to form a cube, the bearings for the 3D far-field implementation is obtained. Instead of generating polar plots, azimuth (theta) is plotted against elevation (phi).

Figure 4.40, Figure 4.41, and Figure 4.42 shows the far-field delay-and-sum beamforming RMS power output for the standard specification, increased temporal sampling frequency, and varying SNR cases respectively. All observations with these variations are the same as in previous sections.

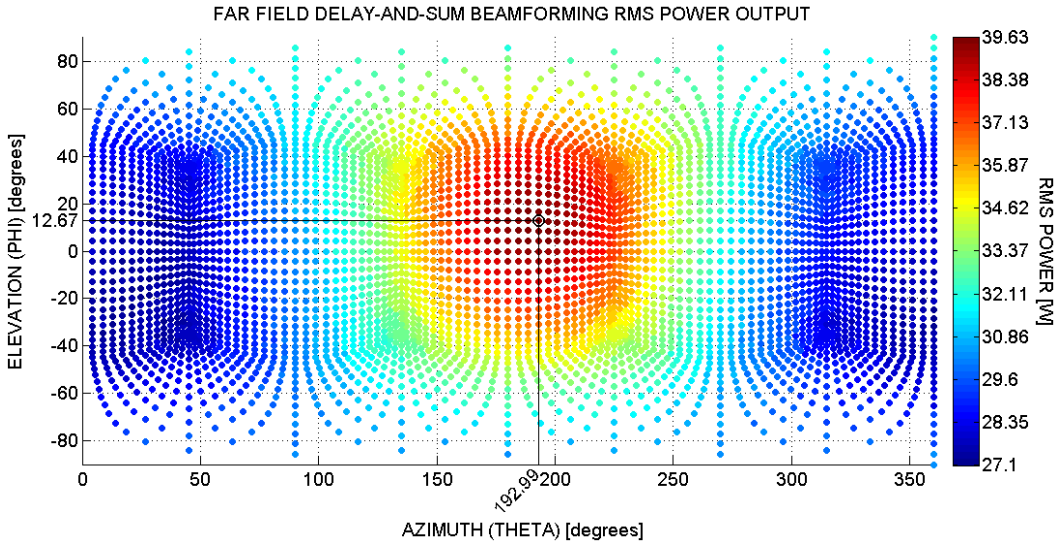

Figure 4.40: Far-field delay-and-sum beamforming RMS power output with standard specifications and source bearing indicated with intersecting lines. Maximum RMS power response beams are indicated with black circles.

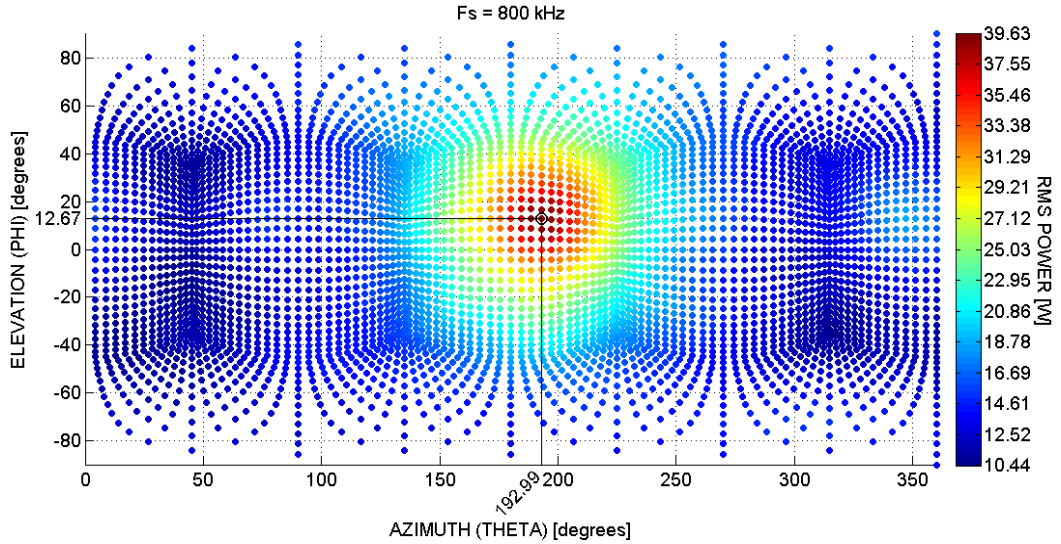

Figure 4.41: Far-field delay-and-sum beamforming RMS power output with temporal sampling frequency increased to 800 kHz and the source bearing indicated with intersecting lines. Maximum RMS power response beams are indicated with black circles.

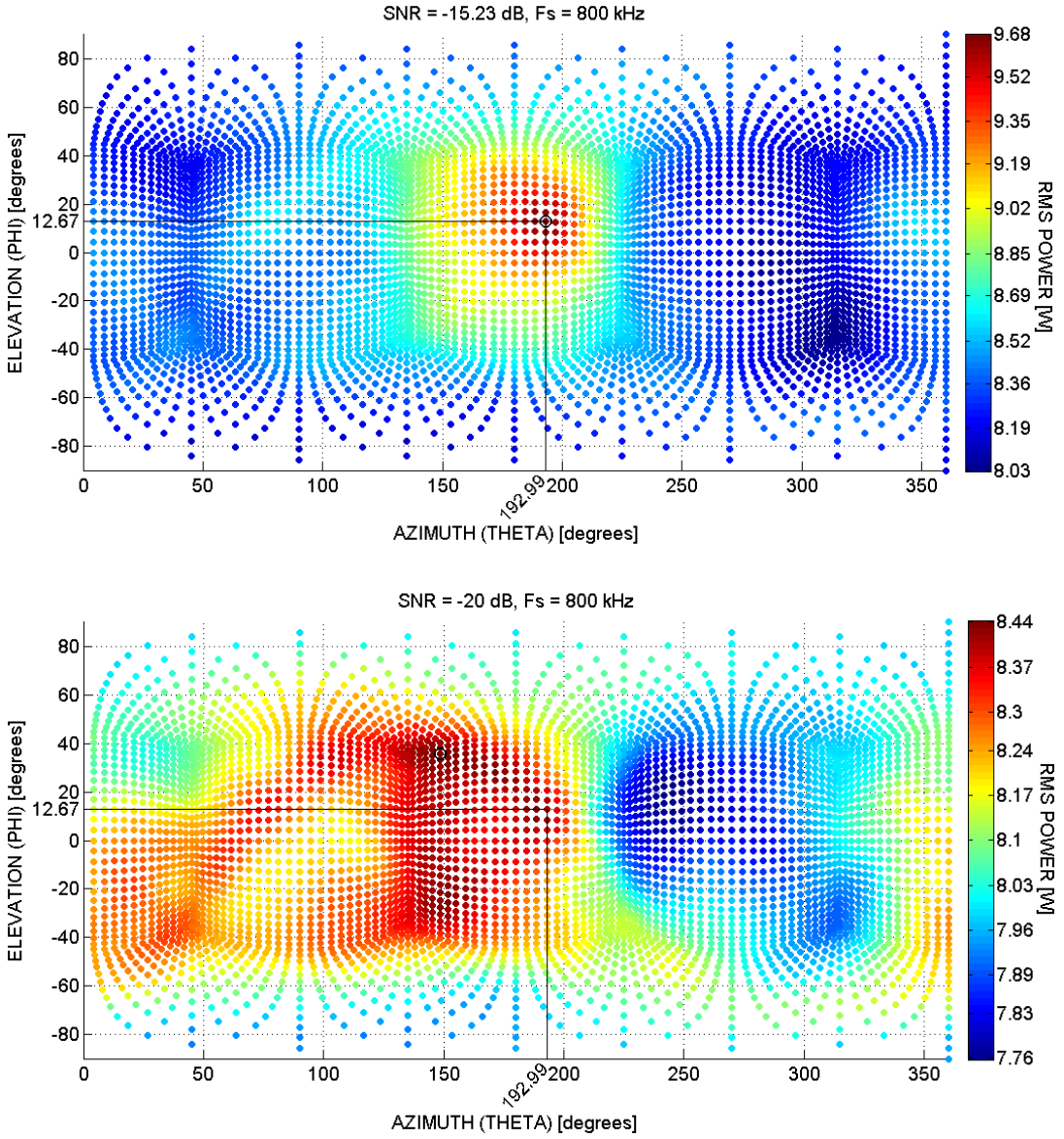

Figure 4.42: Far-field delay-and-sum beamforming RMS power outputs with varying SNR and temporal sampling frequency increased to 800 kHz. The source bearing is indicated with intersecting lines. Maximum RMS power response beams are indicated with black circles.

4.4 Delay-and-Sum Beamformer Simulations in CUDA C

In this section, the results of delay-and-sum beamforming using CUDA C will be presented. Firstly, the test setup with the GPU device used and the chosen parameter values will be presented in section 4.4.1. The CUDA C beamforming program, equivalent to the algorithm used in MATLAB, will then be explained in a sequential manner in section 4.4.2. Snippets relevant to explaining the CUDA C beamforming program will be provided. The complete program code can be found on the CD that accompanies this thesis. Lastly, a performance comparison between pure C and CUDA C will be discussed in section 4.4.3.

Microsoft Visual Studio 2010 and NVIDIA's CUDA toolkit was used for this section's results.

4.4.1 Test Setup

MATLAB was used to fully generate and prepare each of the sensor's signal and delay matrix in the array in the same manner that it was done in previous simulations. These were then exported to text files, used to import the signals and delay matrices into the CUDA C project. The CUDA C program then uses the imported data, with no further changes to it, in order to form all the beams in the grid.

4.4.1.1 GPU Device

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 650"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 2) Multiprocessors x (192) CUDA Cores/MP:    384 CUDA Cores
  GPU Clock rate:                                1072 MHz (1.07 GHz)
  Memory Clock rate:                             2500 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime Version = 5.0, NumDevs = 1, Device0

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0>
```

Figure 4.43: NVIDIA GeForce GTX 650 GPU that is used for CUDA simulations.

Figure 4.43 shows various specifications of the NVIDIA GeForce GTX 650 GPU that is used in the CUDA C simulations. The following parameters are important to the functionality of the CUDA C simulation done in this thesis:

- **CUDA Capability Major/Minor version number:** This version number identifies the features supported by the GPU. It is comprised of a major and a minor revision number where the major revision number indicates the core architecture of the GPU and the minor revision number indicates incremental improvements of this architecture [26]. This GPU has a CUDA capability of 3.0 and is based on the Kepler architecture [26]. A more in depth look at the feature breakdown of each CUDA capability version number can be found in NVIDIA's CUDA C programming guide [26].
- **2 Multiprocessors (MPs) × 192 CUDA Cores/MP:** The grid of blocks passed to a kernel are distributed to the available multiprocessors and CUDA cores at runtime.
- **Maximum number of threads per block:** The maximum number of threads per block in the grid that can be issued to the GPU. This upper limit is important when calling a kernel with its corresponding initialisation values.
- **Concurrent copy and kernel execution:** The concurrent copying of memory from the host to the device or vice versa and the execution of a kernel. This GPU supports it with one copy engine, meaning only one memory transfer and one or more kernel executions can be done concurrently. This feature is dependent on page-locked host memory below.
- **Support host page-locked memory mapping:** As mentioned previously, page-locked host memory guarantees that memory will not be paged out to disk, speeding up memory transfers and providing the ability for concurrent memory transfers and kernel executions.

4.4.1.2 Simulation Parameter Values

The same simulation parameter values that was used to generate Figure 4.26 were used here, with the following deviations from the standard specifications:

- Temporal sampling frequency: $F_s = 1.2\ Mhz$
- With a spacing of two blocks between sensors, the size of each block in the grid needed to satisfy the above spacing between sensors in metres is:

$$G_{size} = \frac{sensor\ spacing\ in\ metres}{sensor\ spacing\ in\ blocks} = \frac{0.075}{2} = 0.0375\ m$$

- Grid resolution: 9 blocks per formed beam
- Array: 4 sensor array as shown in Figure 4.14

4.4.2 The CUDA C Beamformer

In this section, the CUDA C delay-and-sum beamformer will be explained with respect to variable initialisation and memory allocation, the kernel responsible for doing beamforming calculations, and streams.

4.4.2.1 Variable Initialisation and Memory allocation

```
float *Source_Rcvr1 = new float[NUMOFSAMPLES]();
```

Array variable declaration and memory allocation on the host for the signal (source) samples of sensor one. After this declaration, the data for the sensor that was exported from MATLAB are loaded from the text files into this array.

```
float *d_Source_Rcvr1_1;
cudaMalloc((void**)&d_Source_Rcvr1_1, sizeof(float)*NUMOFSAMPLES);
```

Variable declaration and memory allocation on the device for the signal (source) samples of sensor one. *cudaMalloc* is a CUDA library function used to allocate memory on the device.

```
cudaMemcpy(d_Source_Rcvr1_1, Source_Rcvr1, sizeof(float) * NUMOFSAMPLES,
cudaMemcpyHostToDevice);
```

cudaMemcpy is used to copy the contents of a host side variable into the device side variable. The direction of the memory copy (host to device or device to host) is indicated by the *cudaMemcpyHostToDevice* argument. The contents of *Source_Rcvr1* allocated in the host memory now resides in the device memory allocated by *d_Source_Rcvr1_1*.

```
float *d_output1;
cudaMalloc((void**)&d_output1, sizeof(float)*NUMOFSAMPLES);
```

Variable declaration and memory allocation on the device for the output of the CUDA kernel beamformer. A kernel called from the host side cannot return a value as a normal C function can. One solution is to pass a variable as an argument to the CUDA C kernel, store the result of the kernel in the variable, and then copy that variable from the device to the host for further processing.

```
float *h_output1;
cudaHostAlloc((void**)&h_output1, NUMOFSAMPLES*sizeof(float)*81*41,
cudaHostAllocDefault);
```

Variable declaration and page-locked memory allocation on the host for the output of the CUDA kernel beamformer. *cudaHostAlloc* is used to allocate page-locked host memory which enables concurrent memory copies and kernel execution and also has superior performance when compared to global memory. After a kernel has finished executing its tasks, its output contained in *d_output1* will be copied from the device into *h_output1* on the host side.

4.4.2.2 The CUDA Kernel Beamformer

The delay-and-sum beamformer CUDA kernel is used to form one beam partially. The sensors' samples are summed and squared in the kernel. The host then computes the average value of the resulting array, yielding RMS power in the beam.

The specific integer delay in samples needed and the array containing the signal samples of each sensor (before delaying) are passed as arguments to the kernel. Also passed as an argument is the array that will contain the result. The kernel then sums the samples of the four sensors by taking into account the current beam's delay in samples at each sensor and squares each of these summed results. This array is then returned to the host via transferring the array that holds the output. The host then sums the array elements and divides them by the number of elements in the array to produce the power in that beam.

Before explaining the delay-and-sum beamformer kernel that was used for the CUDA C simulations, a simple example will be discussed in order to explain key concepts of a kernel with a focus on functionality that was used in the delay-and-sum beamformer kernel.

Key Concepts

Consider a function in standard C that summates the *N* elements of two arrays (*a* and *b*) and stores the results in a third array (*c*):

```
void add( int *a, int *b, int *c ) {
	for (i=0; i < N; i++)
	{
		c[i] = a[i] + b[i];
	}
}
```

Now consider one possible implementation of the same function in CUDA C:

```
_global_ void add(int *a, int *b, int *c)
{
	int i = threadIdx.x + blockIdx.x * blockDim.x;
	if (i < N)
	{
		c[i] = a[i] + b[i];
	}
}
```

Although these functions look similar, there are a few key differences:

1. **The _global_ qualifier in the kernel**: This qualifier's purpose is to let the compiler know that the kernel can be called from the host and should be executed on the device [25].
2. **The absence of a *for* loop and the declaration of variable *i* in the kernel**: A standard C *for* loop counts from a starting value to a certain maximum value in a sequential manner. Because of this, it does not permit parallel execution of the code that is contained within it. In order to execute the summation of elements *a[i]* and *b[i]* in parallel with *a[i+1]* and *b[i+1]* for example, CUDA makes use of linear thread indices and block indices to define *i*. In the definition of *i*, *threadIdx*, *blockIdx*, and *blockDim* are all declared by the CUDA runtime according to the number of threads per block and the number of blocks in the grid that was specified by the user. *threadIdx* identifies the current linear integer thread index within the block that it resides, *blockIdx* identifies the current linear integer block index in the grid, and *blockDim* identifies the dimensions of the block. If the user declares threads and blocks one-dimensional, only the x dimension of *threadIdx*, *blockIdx*, and *blockDim* are desired. This is denoted by *threadIdx.x* etc.

   The declaration of *i* is thus essentially a linearisation ($i = 0, 1, 2, 3, \ldots$, (*threads per block)* × *(blocks per grid)*) of the thread indices and the block indices that gives each thread that will compute a result a unique linear integer index.

The Delay-and-Sum Beamformer CUDA Kernel

After the thread indices are created in the delay-and-sum beamformer kernel, a number of flags is set in order to sum the elements of each sensor sample array with the next, according to the delay of each sensor corresponding to the current beam.

In MATLAB, the Circshift function was used to delay each sensor's samples with the appropriate amount to form a beam. This function circularly shifts the elements in an array by the specified amount of positions. In CUDA C, this function does not exist and an alternative way had to be created in order to mimic this function and achieve the same results.

For example, after the simulation of the captured signal of each sensor in MATLAB as explained in section 4.2.3 (the signal for each sensor is delayed in reverse by the amount corresponding to the chosen source position), the signal samples for two sensors will look as shown in Table 4.1. In this table, the signal samples does not represent a specific type of signal for ease of explanation. Note that these two sensors are separated by a "distance" of two samples, where sensor 2 are closest to the source.

		Sample Array Index						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Sensor Number	**1**	1,1	1,2	1,3	1,4	1,5	1,6	1,7
	2	1,3	1,4	1,5	1,6	1,7	1,1	1,2

Table 4.1: Two sensor's samples that was simulated as being captured by the array.

After delaying each sensor's samples in MATLAB corresponding to the current beam, in this case matching the actual source position, the samples of the two sensors align as in Table 4.2.

		Sample Array Index						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Sensor Number	**1**	1,1	1,2	1,3	1,4	1,5	1,6	1,7
	2	1,1	1,2	1,3	1,4	1,5	1,6	1,7

Table 4.2: The sensor samples after the beam was formed at the source positon.

In CUDA C, a flag system was used to achieve the same effect. The current linear thread index, the delay (including sign) corresponding to a certain sensor and the current beam, and the number of samples in each sensor's sample array are used to set the flags. Instead of physically shifting each senor's sample array, these flags are then used to sum the elements of the sensor sample arrays at the correct positions corresponding to the delays of the current beam, as if they were physically shifted. Going back to Table 4.1, if the beam at the source position are being formed (the same as in Table 4.2), the following summation will be done after the flags are set:

```
SummedArray[0] = Sensor1[0] + Sensor2[5];
SummedArray[1] = Sensor1[1] + Sensor2[6];
SummedArray[2] = Sensor1[2] + Sensor2[0];
SummedArray[3] = Sensor1[3] + Sensor2[1];
SummedArray[4] = Sensor1[4] + Sensor2[2];
SummedArray[5] = Sensor1[5] + Sensor2[3];
SummedArray[6] = Sensor1[6] + Sensor2[4];
```

This summation gives the same result as the MATLAB implementation but requires no changes in memory. After each summation, the result is squared and saved in the array containing the output. This partially formed beam within the output array is then transferred to the host via a memory copy, where the host then finalises the beam calculation by calculating the beam power after all necessary beams were partially formed on the device. The order in which each kernel call and corresponding memory transfer are done is important and will be discussed in the next section. Refer to Appendix A.1 for a code snippet containing the full code of the delay-and-sum beamformer kernel.

4.4.2.3 Streams

As discussed in section 3.4, streams act as a pipeline of tasks that can execute in parallel with tasks in another stream. This is called task-parallelism. Tasks that are queued in a stream are executed sequentially so that one task must finish before the next task is executed in the stream [25]. The host queues tasks on the device by calling the task with the specified stream as an argument. The device will return control to the host immediately after the task has been queued in the specified stream, allowing the host to queue more tasks on streams while the device is busy executing them [27].

The order in which the host queues tasks in streams are important with respect to performance and the amount of parallelism achieved. An example of two streams will be used to illustrate it. The goal of using two streams in this example is to overlap a memory transfer in one stream with a kernel execution in the other. Each of these streams will hold three tasks: a memory copy from host to device, a kernel launch, and a memory copy from device to host. The first memory copy from host to device includes the data that the kernel needs to perform some operation on. The copy from device to host will hold the result of that operation.

Assuming the device being used has one copy and one kernel execution engine (matching the specifications of the NVIDIA GeForce GTX 650 GPU in Figure 4.43) and that the memory copy and kernel execution has more or less the same execution time, the following execution timeline will occur when all of stream 1’s tasks are queued first, after which all of stream 2’s tasks are queued by the host (the order in which the host queues tasks to the streams are indicated with the number to the left of the task):

	Copy Engine	Kernel Engine
Time ↓	**(1)** Memory copy host to device in **stream 1**	
		(2) Kernel execution in **stream 1**
	(3) Memory copy device to host in **stream 1**	
	(4) Memory copy host to device in **stream 2**	
		(5) Kernel execution in **stream 2**
	(6) Memory copy device to host in **stream 2**	

Table 4.3: Depth-first execution timeline of two streams.

The tasks in Table 4.3 above are queued in each stream *depth-first*, meaning that all tasks are queued in stream 1 before queueing all tasks in stream 2 [25]. Note that the expected concurrent execution between a memory copy of one stream and the kernel execution of another stream does not occur. The reason why this happens is the dependency that exists between the memory copy from device to host and the kernel execution in the same stream. The memory copy from device to host in stream 1 cannot start before the kernel execution in stream 1 has finished and this causes the memory copy to block the copy engine until the kernel execution has finished.

A better approach is to queue the tasks to the streams breadth-first as done in Table 4.4 below [25]:

	Copy Engine	Kernel Engine
Time ↓	**(1)** Memory copy host to device in **stream 1**	
	(2) Memory copy host to device in **stream 2**	**(3)** Kernel execution in **stream 1**
	(5) Memory copy device to host in **stream 1**	**(4)** Kernel execution in **stream 2**
	(6) Memory copy device to host in **stream 2**	

Table 4.4: Breadth first execution timeline of two streams.

Here the kernel execution of stream 1 overlaps with the memory copy from host to device in stream 2. The kernel execution of stream 2 also overlaps with the memory copy from device to host in stream 1.

In the CUDA delay-and-sum beamformer simulations two streams are also used with the intention of overlapping a data transfer in one stream with the kernel execution in another. The host creates two streams in the following manner:

```
cudaStream_t stream1, stream2;
cudaStreamCreate( &stream1 );
cudaStreamCreate( &stream2 );
```

Each of these streams will hold a sequence of two different asynchronous tasks: a kernel call and a memory transfer call to transfer the output of the kernel in the same stream from the device to the host.

The code snippet in Appendix A.2 shows how the CUDA beamformer program iterates over the specified area with a beam that needs to be formed partially at each position in the area's grid. The Beamform kernel execution and a memory copy from device to host are enqueued breadth-first into the two streams. Stream 1 will form a beam at a position and stream 2 will form a beam at the next position in the grid. The output of these kernels are then copied from device to host into two page-locked host memory spaces.

Contained in NVIDIA's CUDA toolkit is a performance profiling application, the NVIDIA Visual Profiler. This application is used to help developers optimise their CUDA applications. The executable program that is generated by Visual Studio are imported into the Visual Profiler. After profiling, an execution timeline is generated that can be used to view the kernel execution and memory copies as they happened in time. To verify that memory copies and kernel execution in the delay-and-sum beamformer simulation are overlapping, a timeline was generated using the Visual Profiler and can be seen in Figure 4.44 below.

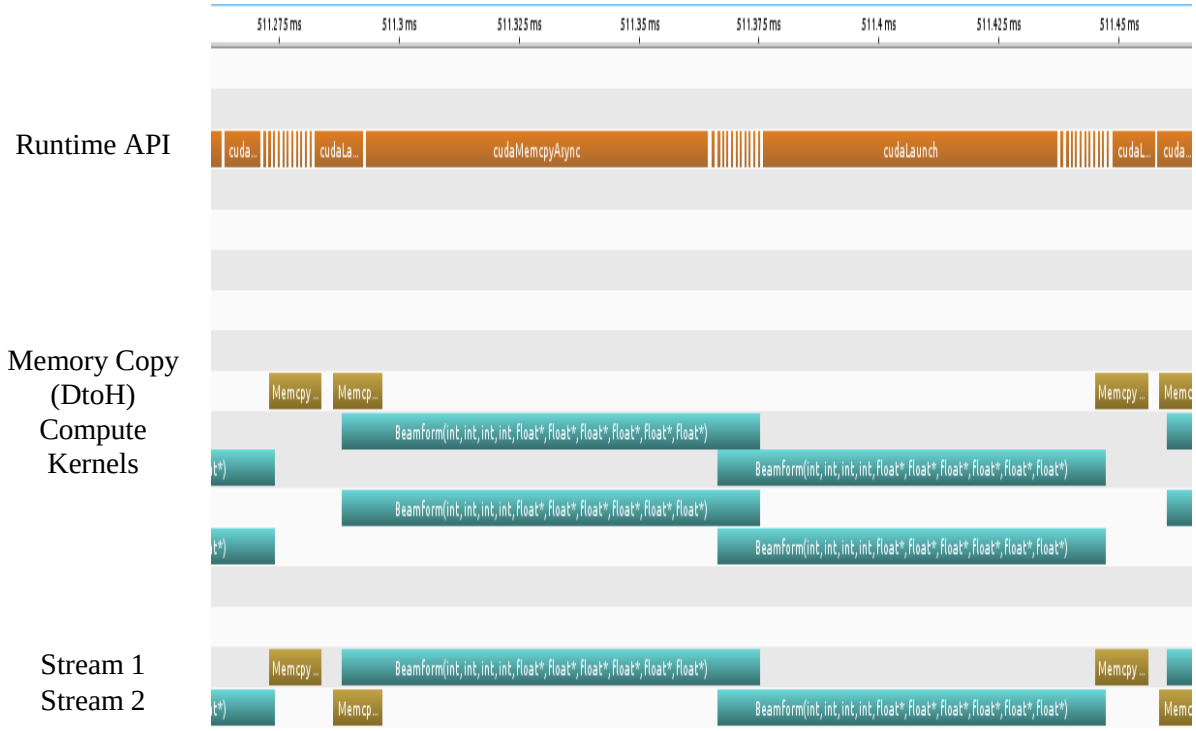

Figure 4.44: NVIDIA Visual Profiler execution timeline of the delay-and-sum beamformer simulation in CUDA.

In this figure, it can be seen that a memory copy in one stream overlaps with kernel execution in another stream. An overlap between kernel execution in one stream also overlaps with kernel execution in another stream. Although these overlaps occur, they are not overlapping as much as expected. The following are possible causes for this:

- **Large amount of per-thread local memory** (as explained in section 3.3) usage. The likelihood that kernels will execute concurrently decreases when large amounts of local memory are used [27].
- There is **only one copy engine** on the device being used. It is clear from the NVIDIA GeForce GTX 650 GPU's device specifications in Figure 4.43 that it only has one copy engine. This prevents concurrent memory copies and explains why there are no overlapping memory copies in the execution timeline above.
- **Dependency checks**, for devices of compute capability 3.0 or lower, prevents memory copies from executing up until all thread blocks of all previous kernel launches from any stream has started executing [26]. All devices have a limited number of threads per multiprocessor, which translates in a limited number of blocks that can be executed on the device at once. Stream 1's memory copy is dependent on the completion of the kernel's execution in the same stream and it is clear from the execution timeline above that this memory copy is waiting for stream 2's kernel to start executing its final collection of thread blocks before it starts with the memory copy.
- The **maximum number of threads per multiprocessor**. Kernels cannot overlap when all threads available are already being used. Only when there are less than the maximum number of threads being used by the kernel currently executing, can the other kernel start executing. This explains why the kernels only start to overlap once the end of the kernel execution in stream 1 are being reached. The solution would be to split each kernel into smaller kernels using fewer threads to execute its operations.

4.4.3 Performance Comparison between CUDA C and Standard C

In order to compare the performance of CUDA C with standard C, the execution time of the delay-and-sum beamformer program are measured for both platforms. The same algorithm that was used in the CUDA C delay-and-sum beamformer simulation was ported to standard C code and can be found on the CD that accompanies this thesis.

The code snippet that is timed is the same for both implementations with the exception that the standard C implementation is not able to execute functions concurrently. These code snippets can be found in Appendix A.2 and A.3. The Personal Computer (PC) that is used to run the standard C code has the following specifications:

- 3.15 GHz Intel Core2 Duo processor with 2 cores, 64 KiB of primary memory cache, and 6144 KiB secondary memory cache.
- 4096 MiB of DDR2 RAM.

The same PC was used to run the CUDA C simulations. The section of the simulation that is being timed is chosen so that the computations execute only on the device when the CUDA C implementation is timed. Enqueueing tasks on the device are the only task done by the host in this chosen section.

The Microsoft Windows API provides high-resolution timers that will be used to measure execution time of both implementations. Firstly, the optimal threads per block combination are found by measuring the execution time in milliseconds of the same code snippet that will be used for the comparison, but with varying threads per block combinations. For each average execution time measurement over 50 iterations, 6561 beams are being formed from 40 000 source samples at each sensor. As mentioned previously in section 3.2, the threads per block must be a multiple of 32 in order to avoid warp divergence. The following results were obtained:

Threads per Block	Average Execution Time over 50 Iterations [ms]
32	1422.35
64	949.42
96	812.1
128	755.96
160	763.97
192	766.01
224	758.11
256	755.77
288	756.39
320	765.71
352	783.58
384	768.6
416	803.53
448	784.43
480	781.05
512	779.53
544	810.43
576	797.5
608	786.05
640	779.18
672	771.85
704	862.69
736	848.86
768	833.76
800	816.55
832	814.17
864	818
896	801
928	795.3
960	787.24
992	800.28
1024	784.54

Table 4.5: Average execution time in CUDA C with varying threads per block.

Overall, 256 threads per block has the fastest execution time and will be used for the performance comparisons to follow.

Table 4.6 shows the results of the average execution time comparison between CUDA C and standard C. The amount of calculations that has to be done to form one beam in the grid are varied by varying the number of samples at each sensor. With 40 000 samples at each sensor the CUDA C, average execution time is a significant 11.67 times faster than the standard C implementation. Notice that this factor increases as the number of samples are increased.

Number of Source Samples at each Sensor	Standard C Average Execution Time over 50 Iterations [ms]	CUDA C Average Execution Time over 50 Iterations [ms]	Factor of Speed Increase from Standard C to CUDA C
10 000	2182.51	388.55	5.62
15 000	3269.42	453.41	7.21
20 000	4360.91	516.38	8.45
25 000	5442.18	578.7	9.4
30 000	6576.75	638.01	10.31
35 000	7630.13	700.69	10.89
40 000	8826.63	756.34	11.67

Table 4.6: Average execution time comparison between CUDA C and standard C.

Figure 4.45 shows a plot of the results above. In this figure, both the standard C and CUDA C results are near-straight lines when plotted, with the standard C results having a much larger gradient than the CUDA C results.

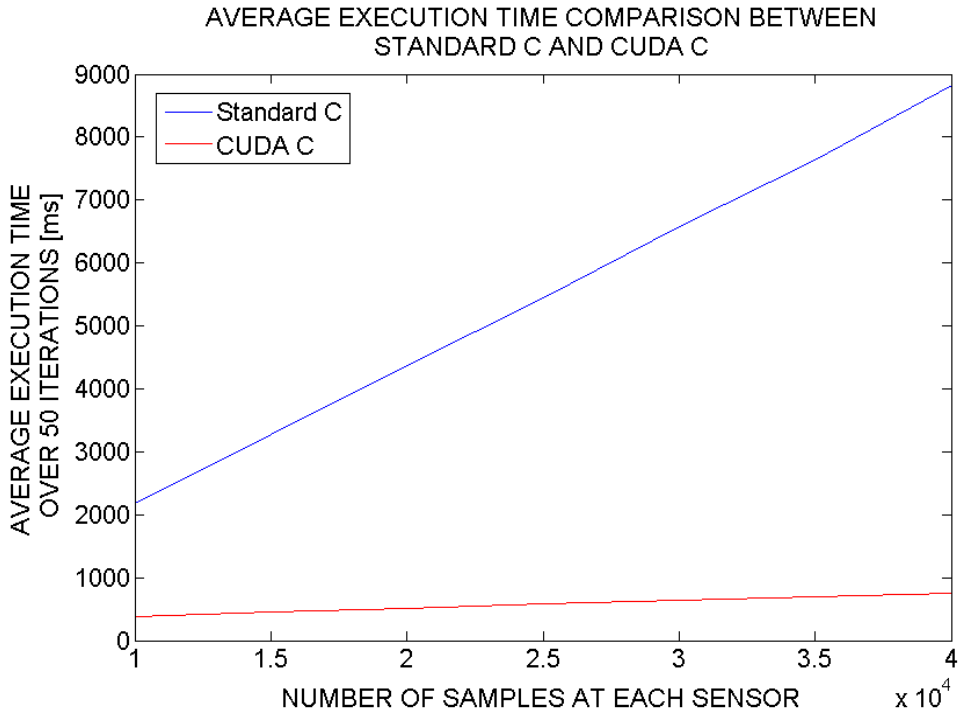

Figure 4.45: Average execution time comparison between CUDA C and Standard C.

The total execution time to complete the simulation for both implementations, timed from where the beams are being formed to the point at which all beams with maximum RMS power response are found, are as follows:

- Standard C: 9816.46 ms
- CUDA C: 1435.56 ms

4.5 Feasibility of a Passive Sonar System on an AUV

To properly determine the feasibility of a passive sonar system on an AUV, in depth theoretical and practical knowledge are needed for a multi-component and complex system such as the passive sonar. This section will serve as a discussion (taking into account the theory and simulations that was presented in this thesis) rather than a determination of the feasibility. Rigid system specifications, the type of AUV used, the types of targets the system needs to detect, the type environment in which the target reside, and more practical testing needs to be done in order to do a proper analysis.

4.5.1 Array

In section 4.3 and 4.4 meaningful delay-and-sum beamforming was achieved of signals with an upper frequency of 10 kHz in the signal frequency band using an array that contains a small amount of sensors. Given the maximum frequency in the signal frequency band, the size of the arrays used were comparatively small with respect to the size of an AUV.

The type of target that needs to be detected, the maximum distance of the target, and the environment that the target resides in mostly determines the type of array needed. Array specifications that are determined by these factors are the spacing between sensors (to avoid spatial aliasing), the required resolution of the array that is acceptable for the type of target, and the amount of sensors needed (array gain needed to suppress noise in the environment, worsening with increasing distance). Together, sensor spacing and the amount of sensors needed determines array size.

The type of vehicle they are mounted on determines many arrays' configurations. For a small vehicle such as an AUV, a retractable array (similar to Figure 2.28) seems like the best choice, with the following benefits:

- Array size is not determined by the spacing needed between sensors when the array is retractable.
- The distance between sensors can be controlled, enabling operation at different frequencies.
- When the array is fully retracted, the draught caused by the array when the vehicle is moving through the water is minimised, which in turn minimises the battery power consumption needed for propulsion (if an electric motor is used).

4.5.2 Power Consumption

Without any rigid system specifications, acceptable power consumption of the electronic systems used by the passive sonar cannot be determined. The processing unit of a passive sonar system plays a big role in overall power consumption and mobile processors with embedded CUDA capable GPU's (such as the NVIDIA Tegra X1) and Field-Programmable Gate Arrays (FPGA's) both offer power consumption under 10W with significant improvements in power consumption expected for GPU's in the coming years [28] [29].

4.5.3 Computational Power

Without any rigid system specifications, the computational power needed cannot be determined. However, from section 4.4 and the ongoing improvements in computational power per Watt of GPUs and FPGAs [28], it seems unlikely that the computational power needed by a passive sonar system on an AUV will be a hindrance.

4.5.4 Application

Limits imposed by the AUV on the acceptable dimensions of the array and power consumption reduces the types of applications of a passive sonar used on an AUV drastically. With the array sensor spacing determined by the upper frequency of the band in which the target's signal that needs to be detected resides, the amount of sensors in an array with a constrained size reduces as the upper frequency lowers and consequently array gain as well. Thus, the targets that the passive sonar needs to detect should preferably have a high as possible upper frequency in the target's signal band. Unfortunately, in a fluid medium, increasing frequency also translates into stronger signal attenuation with distance.

The detection of divers near a port, naval base, or sea faring vessel using a sonar is a fairly new topic and sonars used for this purpose are called Diver Detection Sonars (DDSs) [5]. Ports and naval bases can be vulnerable to an underwater attack of this kind because of the complex issue of securing these areas with underwater fences, as the fence itself also needs to be protected and monitored. The constructions of these fences are also difficult due to it being dependant on the geography of the bottom of the sea. DDSs can be active, where the body of the diver is used for detection, or passive, where the noise of the breathing system used by the diver is used for detection [5].

The uppermost frequency in the signal frequency band of a diver's breathing system is approximately 80 kHz with a bandwidth of around 45 kHz, requiring an array sensor spacing that is short in comparison with the size of an AUV [30]. Physical limitations imposed by the diver's body reduces the need for very long distance detection of the diver. This also reduces the array size needed (to perform near-field beamforming at large distances where the position of the diver can be estimated, a large array is needed) and reduces the number of sensors needed in the array (array gain needed to detect the diver at distance).

The abovementioned statements make the use of a passive sonar system on an AUV for diver detection seem like the most viable application thereof, but does not prove its viability and further investigation into the technical aspects of it and the need for such a system needs to be done. Given that these aspects are viable, an AUV with a passive sonar system could be used to patrol shorelines where other security measures such as statically mounted DDS systems are not feasible. Envisioned is such an AUV that patrols a certain area by visiting checkpoints. At each checkpoint, the AUV will land on the sea floor or be kept stationary in space, deploy its retractable array (the same approach as in [19]), collect data, and move on to the next checkpoint.

Chapter 5 Conclusion and Recommendations for Future Work

5.1 Conclusions

In this project the theory of a passive sonar system, simulations in two and three dimensions of the digital delay-and-sum beamforming algorithm in MATLAB and CUDA C, and the feasibility of a passive sonar system on an AUV were discussed.

From the theory researched in this project, it was derived that the array that a passive sonar system uses can quickly become too large for application on an AUV. The number of sensors in the array translates into a higher array gain (yielding higher SNR), and a narrower beam width in the principle direction of reception. The sensor spacing should be kept at $d \leq \lambda/2$ to avoid spatial aliasing and grating lobes from appearing in the beam pattern of the array. Consequently, depending on the SNR required for target detection by the desired type of application of the passive sonar, the number of sensors in the array and the array spacing needed can make the dimensions of the array large. It was also shown that the dimensions of the array determines the extent of the near-field of the array where the exact position of the target can be estimated. However, the need for exact target position estimation depends on the application of the passive sonar system.

As a basis to work from, the signals of a submarine at periscope depth and ambient noise commonly found in the sea was simulated with the use of MATLAB. Using these signals, a near-field and far-field delay-and-sum beamformer in two and three dimensions was simulated. It was shown that meaningful results could be obtained by using small arrays that contain four or nine sensors in two dimensions and eight sensors in three dimensions. Only small areas in the near-field beamforming case was investigated with a maximum source distance of 10.33m. Increasing the number of synchronous beams via increasing the temporal sampling frequency had significant beneficial effects of identifying source position or bearing. Resilience against decreasing SNR was also shown to improve when the number of synchronous beams were increased. Decreasing the number of beams formed in the area being scanned reduces ambiguity of the source position but compromises the beamformer resolution.

The learning curve of implementing a delay-and-sum beamformer in CUDA C proved to be steep but short when prior knowledge of C exists. The use of pinned CPU memory and CUDA streams facilitated concurrent execution of memory copies (between CPU and GPU memory) and kernel executions on the GPU. Concurrent execution of kernels were also achieved. NVIDIA's GeForce GTX 650 GPU and a 3.15 GHz Intel Core2 Duo CPU was used to do a performance comparison between CUDA C and standard C. The CUDA C code executed on the GPU had a maximum of an 11.67-fold decrease in execution time than the standard C code running on the CPU.

The use of a retractable array overcomes the restrictions of array size that is imposed by the body of an AUV. It also facilitates multiple operating frequencies whilst avoiding spatial aliasing by being able to control sensor spacing. With the ongoing improvements in both

computational power and power consumption of FPGAs and mobile processors with embedded GPUs, it seems that these two factors will not be a hindrance for a passive sonar system on an AUV. Again, the type of application will ultimately determine this. The absence of the need for long distance near-field detection combined with an upper frequency of 80 kHz of the signal emitted by a diver's breathing system (requiring short sensor spacing) makes a diver detection passive sonar seem like the most feasible application of a passive sonar on an AUV.

5.2 Recommendations for Future Work

Each element in a passive sonar system has proven to be easily understandable on the surface, although deeper understanding of each element, all its possible implementations, and the theory that govern it can become complex. This project builds a good foundation for understanding the passive sonar system, although much further development is needed.

5.2.1 Theoretical Recommendations

Not all of the theory that was covered in Chapter 2 were implemented in the delay-and-sum beamformer simulations. Looking at beams with maximum RMS power response was an effective way to observe the effects of changing parameters of the delay-and-sum beamformer. This is by no means a practical solution to determine the source position for a non-ideal environment. Detection theory should thus be implemented in the simulations, which will enable measurement of system performance. Array shading should also be implemented to improve the beam pattern of the array, an optimisation that is especially important for arrays with a small amount of sensors.

Ideal ambient noise (homogeneous and isotropic) were considered in this project. Ambient noise found in the sea will have significant detrimental effects on the detection of targets. An understanding of these effects are needed for proper system design decisions.

For a certain array, it was shown in Chapter 2 that primitive assumptions on the distance at which far-field beamforming can be applied can be erroneous. The error in signal delay to form a beam should rather be measured at different steering angles and distances. An acceptable error in signal delay at each sensor to form a certain beam should then be chosen, facilitating the calculation of accurate distances for each angle at which far-field beamforming can be applied.

Depending on the resolution of the beamformer, beamforming in the array's near-field can pose extreme computational cost when scanning a large area. By firstly scanning an area more coarsely by reducing the amount of beams formed and then scanning the areas likely to contain the source position, this problem can be alleviated. This method will also reduce the overall amount of beams that needs to be formed to find a source position in an area.

In section 4.4 beams were partially formed with a CUDA kernel by summing the data of the sensors and squaring it. Using the result, the CPU calculated the average of the resulting floating-point value array, yielding the RMS power. Array reduction is the process of taking an input array and performing computations on it, with the resulting array containing fewer values. The average that the CPU calculated can be done using CUDA and array reduction.

Essentially, the array is split up into smaller parts where each part is then summed using a thread. This process is then repeated until only one value remain, equal the sum of all the values in the array. Array reduction will decrease execution time of the CUDA C delay-and-sum beamformer even further.

5.2.2 System Recommendations

A passive sonar system with all of its components can become expensive. Seeing that the feasibility of a passive sonar system on an AUV cannot be fully determined yet, concept testing via using real life collected data and simulations can facilitate a much deeper understanding of all the complexities of a passive sonar system on an AUV. This is especially true with respect to the design of the recommended retractable array and the feasibility of different types of applications of such a system.

The MAX1308 Evaluation Kit

The MAX1308 Evaluation Kit is a complete data-acquisition system and was acquired in order to sample data of an 8-sensor array simultaneously, an important requirement needed to preserve the phase information between the data captured by the sensors of the array. It comes with software, requiring no development time. The kit uses a MAX1308 ADC. This 12-bit ADC can sample eight inputs simultaneously at a maximum frequency of 456 kHz per channel. Unfortunately, a combination of time constraints and a change in thesis focus prohibited the use of this device to capture real life data.

This device can be used to effortlessly capture real life data of an array in different environments. The data can then be used as input to beamforming simulations, making it a simple process to test the feasibility of different system applications, array designs, the effects of different environments, and different algorithm implementations.

As mentioned previously, a diver detection passive sonar is a good candidate for application of a passive sonar on an AUV. This application recommendation can be tested using the data captured underwater of a diver's breathing system.

FPGA versus GPU and CUDA as processing unit

This thesis focussed on parallel processing using a GPU and CUDA. Although results were promising, further investigation with respect to the best solution for a processing unit for parallel computations on an AUV needs to be done. FPGAs and GPUs are technologies that are commonly considered for intensive parallel processing tasks. Advantages of using FPGAs over GPUs include processing power per Watt, deterministic timing, and flexible interfacing [28]. Similarly for GPUs, FLOPS, development time, backward compatibility, and processing power per cost [28].

Chapter 6 Bibliography

[1] R. J. Urick, Principles of Underwater Sound for Engineers, New York: McGraw-Hill, 1967.

[2] A. A. Winder, "Sonar System Technology," *IEEE Transactions on Sonics and Ultrasonics,* Vols. su-22, no. 5, pp. 291-332, 1975.

[3] H. G. Urban, Handbook of Underwater Acoustic Engineering, Bremen: STN ATLAS Elektronik GmbH, 2002.

[4] Z. M. X. Walter, J. Harwood, P. L. Tyack, M. P. Johnson and M. T. Peter, "Passive Acoustic Detection of Deep-Diving Beaked Whales," *Acoustical Society of America,* vol. 124, no. 5, pp. 2823-2832, 2008.

[5] Q. Li, Digital Sonar Design in Underwater Acoustics, Hangzhou: Zhejiang University Press, Springer, 2012.

[6] D. H. Johnson and D. E. Dudgeon, Array Signal Processing: Concepts and Techniques, Englewood Cliffs: P T R Prentice Hall, 1993.

[7] C. H. Sherman and J. L. Butler, Transducers and Arrays for Underwater Sound, New York: Springer Science & Business Media, 2007.

[8] R. P. Hodges, Underwater Acoustics: Analysis, Design and Performance of Sonar, Chichester: John Wiley & Sons Ltd, 2010.

[9] W. A. D., Sonar for Pracitising Engineers Third Edition, Chichester: John Wiley & Sons Ltd, 2002.

[10] L. S. Lasdon, A. D. Waren and D. Suchman, "Optimal Design of Acoustic Sonar Transducer Arrays," *Engineering in the Ocean Environment,* vol. 2, pp. 3-9, 1974.

[11] S. N. Khasim, M. Y. Krishna, J. Thati and V. M. Subbarao, "Analysis of Different Tapering Techniques for Efficient Radiation Pattern," *e-Journal of Science & Technology,* vol. 8, no. 5, pp. 47-53, 2013.

[12] D. M. Pozar, Microwave and RF Design of Wireless Systems, New York: John Wiley & Sons, Inc., 2001.

[13] J. G. Proakis and D. G. Manolakis, Digital Signal Processing: Principles, Algorithms, and Applications, Upper Saddle River, New Jearsey: Pearson Prentice Hall, 2007.

[14] M. F. Duarte, "OpenStax CNX," 14 August 2013. [Online]. Available: http://cnx.org/contents/38b50425-5eaa-4dd1-9589-57b9d7488da0@3/Aliasing-Phenomena. [Accessed 22 11 2015].

[15] R. Scholte, B. Roozen and I. Lopez, "On Spatial Sampling and Aliasing in Acoustic Imaging," in *Twelfth International Congress on Sound and Vibration*, Lisbon, 2005.

[16] J. G. Ryan, "Criterion for the minimum source distance at which plane-wave beamforming can be applied," *Journal of Acoustical Society of America,* vol. 104, no. 1, pp. 595-598, 1998.

[17] J.-P. Marage and Y. Mori, Sonar and Underwater Acoustics, New York: John Wiley & Sons, Inc., 2010.

[18] L. Josefsson and P. Persson, Conformal Array Antenna Theory and Design, New Jersey: John Wiley & Sons, Inc., 2006.

[19] S. A. L. Glegg, M. . P. Olivieri, R. . K. Coulson and S. M. Smith, "A Passive Sonar System Based on an Autonomous Underwater Vehicle," *IEEE Journal of Oceanic Engineering*, vol. 26, no. 4, pp. 700-710, 2001.

[20] W. C. Knight, R. G. Pridham and S. M. Kay, "Digital Signal Processing for Sonar," in *Proceedings of the IEEE*, 1981.

[21] A. Dr. Greensted, "The Lab Book Pages," 27 October 2012. [Online]. Available: http://www.labbookpages.co.uk/audio/beamforming/delaySum.html. [Accessed 12 12 2014].

[22] R. G. Pridham and R. A. Mucci, "A Novel Approach to Digital Beamforming," *Journal of Acoustic Society America,* vol. 63, no. 2, pp. 425-434, 1978.

[23] R. L. Dawe, "Detection Threshold Modelling Explained," DSTO Aeronautical and Maritime Research Laboratory, Melbourne, 1997.

[24] A. D. Waite, Sonar for Practising Engineers Third Edition, Chichester: John Wiley & Sons Ltd, 2002.

[25] J. Sanders and E. Kandrot, CUDA by Example, An Introduction to General-Purpose GPU Programming, Boston: Addison-Wesley, 2011.

[26] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide v6.5,* Santa Clara: NVIDIA Corporation, 2014.

[27] NVIDIA Corporation, NVIDIA CUDA C Programming Guide v4.2, Santa Clara: NVIDIA Corporation, 2012.

[28] BERTEN, “GPU vs FPGA Performance Comparison,” BERTEN DSP S.L., Santander, 2016.

[29] NVIDIA, “GPU-Based Deep Learning Inference: A Performance and Power Analysis,” NVIDIA, Santa Clara, 2015.

[30] K. W. Lo and B. G. Ferguson, “Diver Detection and Localization Using Passive Sonar,” *Proceedings of Acoustics* , 2012.

Appendix A
Code Snippets

A.1 Delay-and-sum Beamformer Kernel

```
// The CUDA kernel responsible for forming one delay and sum beam and to calculate the
current beam RMS power response partially by squaring each element.
__global__ void Beamform(int delay1, int delay2, int delay3, int delay4, float
*d_Source_Rcvr1, float *d_Source_Rcvr2, float *d_Source_Rcvr3, float *d_Source_Rcvr4,
float *d_output)
{
	int linID = threadIdx.x + blockIdx.x * blockDim.x;

	// Flags for each receiver that is used to mimic circular shifting of an array
	equivalent to Matlab's circshift() function.
	int flag_plus_minus1 = 0, flag_plus_minus2 = 0, flag_plus_minus3 = 0,
	flag_plus_minus4 = 0, flag_boundary1 = 0, flag_boundary2 = 0, flag_boundary3 =
	0, flag_boundary4 = 0;

	if (delay1 > 0)
	{
		flag_plus_minus1 = 1;
		flag_boundary1 = 1;
	}
	else
	{
		flag_plus_minus1 = -1;
	}
	if (delay2 > 0)
	{
		flag_plus_minus2 = 1;
		flag_boundary2 = 1;
	}
	else
	{
		flag_plus_minus2 = -1;
	}
	if (delay3 > 0)
	{
		flag_plus_minus3 = 1;
		flag_boundary3 = 1;
	}
	else
	{
		flag_plus_minus3 = -1;
	}
	if (delay4 > 0)
	{
		flag_plus_minus4 = 1;
		flag_boundary4 = 1;
	}
	else
	{
		flag_plus_minus4 = -1;
	}

	if (linID < NUMOFSAMPLES)
	{
```

```
                if ((linID - (delay1) + (NUMOFSAMPLES)*flag_plus_minus1*flag_boundary1)
                >= NUMOFSAMPLES)
                {
                        if (flag_boundary1 == 0)
                        {
                                flag_boundary1 = 1;
                        }
                        else
                        {
                                flag_boundary1 = 0;
                        }
                }
                if ((linID - (delay2) + (NUMOFSAMPLES)*flag_plus_minus2*flag_boundary2)
                >= NUMOFSAMPLES)
                {
                        if (flag_boundary2 == 0)
                        {
                                flag_boundary2 = 1;
                        }
                        else
                        {
                                flag_boundary2 = 0;
                        }
                }
                if ((linID - (delay3) + (NUMOFSAMPLES)*flag_plus_minus3*flag_boundary3)
                >= NUMOFSAMPLES)
                {
                        if (flag_boundary3 == 0)
                        {
                                flag_boundary3 = 1;
                        }
                        else
                        {
                                flag_boundary3 = 0;
                        }
                }
                if ((linID - (delay4) + (NUMOFSAMPLES)*flag_plus_minus4*flag_boundary4)
                >= NUMOFSAMPLES)
                {
                        if (flag_boundary4 == 0)
                        {
                                flag_boundary4 = 1;
                        }
                        else
                        {
                                flag_boundary4 = 0;
                        }
                }

                d_output[linID] = pow(d_Source_Rcvr1[linID - (delay1) + (NUMOFSAMPLES) *
                flag_plus_minus1 * flag_boundary1]
                + d_Source_Rcvr2[linID - (delay2) + (NUMOFSAMPLES) * flag_plus_minus2 *
                flag_boundary2]
                + d_Source_Rcvr3[linID - (delay3) + (NUMOFSAMPLES) * flag_plus_minus3 *
                flag_boundary3]
                + d_Source_Rcvr4[linID - (delay4) + (NUMOFSAMPLES) * flag_plus_minus4 *
                flag_boundary4], 2);

        }
}
```

A.2 Queuing of Tasks in Streams by the Host

```
for (int row = 0; row<GRIDSIZE; row++)
{
	for (int col = 0; col<GRIDSIZE; col+=2)
	{

		Beamform<<<blocks, threads, 0, stream1>>>(DM_Rcvr1[row][col],
		DM_Rcvr2[row][col],
		DM_Rcvr3[row][col],
		DM_Rcvr4[row][col],
		d_Source_Rcvr1_1,
		d_Source_Rcvr2_1,
		d_Source_Rcvr3_1,
		d_Source_Rcvr4_1,
		d_output1);

		if((col+1) < GRIDSIZE)
		{
			Beamform<<<blocks, threads, 0, stream2>>>(DM_Rcvr1[row][col+1],
			DM_Rcvr2[row][col+1],
			DM_Rcvr3[row][col+1],
			DM_Rcvr4[row][col+1],
			d_Source_Rcvr1_2,
			d_Source_Rcvr2_2,
			d_Source_Rcvr3_2,
			d_Source_Rcvr4_2,
			d_output2);
		}

		cudaMemcpyAsync(h_output1 + counter2,
		d_output1,
		sizeof(float)*NUMOFSAMPLES,
		cudaMemcpyDeviceToHost,
		stream1);

		if((col+1) < GRIDSIZE)
		{
			cudaMemcpyAsync( h_output2 + counter2,
			d_output2,
			sizeof(float)*NUMOFSAMPLES,
			cudaMemcpyDeviceToHost,
			stream2 );
		}

		counter1+=2;
		counter2+=NUMOFSAMPLES;
	}
}
```

A.3 Standard C Implementation of Delay-and-Sum beamformer

```
for (int row = 0; row<GRIDSIZE; row++)
{
	for (int col = 0; col<GRIDSIZE; col++)
	{
		// Get the delay in samples for each receiver signal needed to form the
		beam.
		delay1 = DM_Rcvr1[row][col];
		delay2 = DM_Rcvr2[row][col];
		delay3 = DM_Rcvr3[row][col];
		delay4 = DM_Rcvr4[row][col];

		for (int linID = 0; linID<NUMOFSAMPLES; linID++)
		{
			// Set flags to mimic circular "shifting" of the signal of each
			receiver.
			if (delay1 > 0)
			{
				flag_plus_minus1 = 1;
				flag_boundary1 = 1;
			}
			else
			{
				flag_plus_minus1 = -1;
				flag_boundary1 = 0;
			}

			if (delay2 > 0)
			{
				flag_plus_minus2 = 1;
				flag_boundary2 = 1;
			}
			else
			{
				flag_plus_minus2 = -1;
				flag_boundary2 = 0;
			}

			if (delay3 > 0)
			{
				flag_plus_minus3 = 1;
				flag_boundary3 = 1;
			}
			else
			{
				flag_plus_minus3 = -1;
				flag_boundary3 = 0;
			}

			if (delay4 > 0)
			{
				flag_plus_minus4 = 1;
				flag_boundary4 = 1;
			}
			else
			{
				flag_plus_minus4 = -1;
				flag_boundary4 = 0;
			}
```

```
if (linID < NUMOFSAMPLES)
{
	if ((linID - (delay1) +
	(NUMOFSAMPLES)*flag_plus_minus1*flag_boundary1) >=
	NUMOFSAMPLES)
	{
		if (flag_boundary1 == 0)
		{
			flag_boundary1 = 1;
		}
		else
		{
			flag_boundary1 = 0;
		}
	}
	if ((linID - (delay2) +
	(NUMOFSAMPLES)*flag_plus_minus2*flag_boundary2) >=
	NUMOFSAMPLES)
	{
		if (flag_boundary2 == 0)
		{
			flag_boundary2 = 1;
		}
		else
		{
			flag_boundary2 = 0;
		}
	}
	if ((linID - (delay3) +
	(NUMOFSAMPLES)*flag_plus_minus3*flag_boundary3) >=
	NUMOFSAMPLES)
	{
		if (flag_boundary3 == 0)
		{
			flag_boundary3 = 1;
		}
		else
		{
			flag_boundary3 = 0;
		}
	}
	if ((linID - (delay4) +
	(NUMOFSAMPLES)*flag_plus_minus4*flag_boundary4) >=
	NUMOFSAMPLES)
	{
		if (flag_boundary4 == 0)
		{
			flag_boundary4 = 1;
		}
		else
		{
			flag_boundary4 = 0;
		}
	}

	// Form one delay and sum beam and calculate the current
	beam RMS power response partially by squaring each
	element.
	// Each beam RMS power response will be NUMOFSAMPLES long
	and summation and division by NUMOFSAMPLES to
```

```
                        // get the final RMS power response is done later in order
                        to be able to properly compare execution time with CUDA C
                        implemenation.
                        output1[beamCounter + linID] = pow(Source_Rcvr1[linID -
                        (delay1) + (NUMOFSAMPLES)*flag_plus_minus1*flag_boundary1]
                        + Source_Rcvr2[linID - (delay2) +
                        (NUMOFSAMPLES)*flag_plus_minus2*flag_boundary2]
                        + Source_Rcvr3[linID - (delay3) +
                        (NUMOFSAMPLES)*flag_plus_minus3*flag_boundary3]
                        + Source_Rcvr4[linID - (delay4) +
                        (NUMOFSAMPLES)*flag_plus_minus4*flag_boundary4], 2);

                }
            }
            beamCounter += NUMOFSAMPLES;
        }
}
```